Lecture Notes in Computer Science 2269
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Stephan Diehl (Ed.)

Software Visualization

International Seminar
Dagstuhl Castle, Germany, May 20-25, 2001
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Stephan Diehl
Saarland University, FR Informatik
PO Box 15 11 50, 66041 Saarbrücken, Germany
E-mail: diehl@acm.org

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Software visualization : international seminar, Dagstuhl castle, Germany,
May 20 - 25, 2001 ; revised papers / Stephan Diehl (ed.). - Berlin ;
Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ;
Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2269)
ISBN 3-540-43323-6

CR Subject Classification (1998):D.2, D.1.7, F.2, G.2, H.2, I.3

ISSN 0302-9743
ISBN 3-540-43323-6 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10846115 06/3142 5 4 3 2 1 0

Preface

Researchers in software visualization develop and investigate methods and use of computer-graphical representations of different aspects of software such as its static structure, its concrete or abstract execution as well as its design and evolution.

Since Goldstein's and von Neumann's demonstration of the usefulness of flowcharts in 1947 visual representations have played an important role in understanding and designing programs. Software visualization is done in many areas of computer science, but often not realized as a field of its own. As a result papers are published at conferences and workshops of these areas reinventing the wheel over and over again.

The planning for this book started at the Dagstuhl Seminar on Software Visualization during May 2001. The goal of this seminar was to bring together practitioners and researchers working in the area of software visualization as well as those working in related areas including database visualization, graph drawing, and visual programming. Discussions and presentations at the seminar were not restricted to theoretical foundations and technical applications. They also addressed psychological and educational aspects.

The intent of this book is to present the state of the art in software visualization. To this aim it contains introductory papers and original work. More than 60 authors contributed to this volume. It is divided into five chapters:

- algorithm animation,
- software visualization and software engineering,
- software visualization and education,
- graphs in software visualization,
- and perspectives of software visualization.

Each chapter starts with an introduction surveying previous and current work and providing extensive bibliographies.

Eventually we hope that this volume will foster software visualization and its impact on the way we teach, learn, and design programs.

The editor would like to gratefully acknowledge the support provided by the authors, Springer-Verlag, and the Dagstuhl staff during the seminar and the making of this volume.

December 2001 Stephan Diehl

Foreword by the Organizers

The International Dagstuhl Seminar on Software Visualization was held in May 2001 at the International Research and Conference Center for Computer Science in Schloss Dagstuhl, Germany. Dagstuhl seminars are one-week meetings which bring together the most significant researchers on important topics in computer science. Participation is by invitation only.

It is often said that humans have never before created any artifacts which are as complex as today's software systems. As a result creating, maintaining, understanding, and teaching software is a challenging task. Software is neither matter nor energy, it is just a kind of information. Sometimes the representation and the information itself are confused. Software visualization is concerned with visually representing different aspects of software including its structure, execution, and evolution. So far, research on software visualization has mostly been motivated by its potential to support teaching. Many systems have been developed to facilitate the production of algorithm animations. At Dagstuhl software engineers and re-engineers have repeatedly argued that there is a strong need for software visualization in their areas. Here further research includes the use of techniques from information visualization to display software metrics, graph layout and graph animations to show the structure and changes in software systems, and program animation for debugging. At the seminar more than 50 researchers from all around the world discussed the state of the art as well as challenging questions for the future of software visualization. The program included 38 presentations and 15 system demonstrations, as well as several sessions for group discussions. Participants of the seminar volunteered

- to compile a post seminar proceedings, which is to be published as a Springer LNCS state-of-the-art survey,
- to create a repository with algorithm animations, and software visualization tools
- to initiate an international conference series on software visualization.

We feel that the seminar was a seminal event. The future will tell whether it reached its ambitious goals to form a community and raise awareness of software visualization as a challenging and important research field of its own.

December 2001

Stephan Diehl
Peter Eades
John Stasko

Table of Contents

Chapter 1
Algorithm Animation

Introduction

Andreas Kerren[1] and John T. Stasko[2]

[1] FR 6.2 Informatik,
Saarland University,
PO Box 15 11 50, D-66041 Saarbrücken, Germany.
kerren@cs.uni-sb.de,
http://www.cs.uni-sb.de/~kerren/

[2] College of Computing / GVU Center,
Georgia Institute of Technology,
Atlanta, GA 30332-0280, USA.
stasko@cc.gatech.edu,
http://www.cc.gatech.edu/~john.stasko/

An *algorithm animation* (AA) visualizes the behavior of an algorithm by producing an abstraction of both the data and the operations of the algorithm. Initially it maps the current state of the algorithm into an image, which then is animated based on the operations between two succeeding states in the algorithm execution. Animating an algorithm allows for better understanding of the inner workings of the algorithm, furthermore it makes apparent its deficiencies and advantages thus allowing for further optimization.

Price, Baecker and Small [63] distinguish between algorithm animation and program animation. The first term refers to a dynamic visualization of the higher-level descriptions of software (algorithms) that are later implemented in software. The second term refers to the use of dynamic visualization techniques to enhance human understanding of the actual implementation of programs or data structures. Price, Baecker, and Small define both areas of study to collectively be a part of *Software Visualization* (SV). Here in this introduction we loosen this distinction, i.e., the discussed systems can be subsumed by the terms algorithm and program animation.

Two extensive anthologies about software visualization were published in 1996 and 1998 [33,78]. Both provide overviews of the field. The latter one also contains revised versions of some seminal papers on classical algorithm animation systems as well as educational and design topics. Other published articles provide summaries of different aspects of algorithm animation in particular, including taxonomies [10], the use of abstraction [22], and user interface issues [39]. In this introduction we first provide a short summary of the historical development of software visualization, especially algorithm and program visualization. In this context we concentrate on systems that introduced new concepts.

Next, we survey some newer systems on the basis of four concepts or dimensions: specification technique, visualization technique, language paradigm, and domain specific animations. Because these systems have been developed more recently, they usually were not discussed in the aforementioned anthologies. Due to space limitations, however, the following sections only describe some representative systems, with a particular focus on systems that are presented in the seven papers of this chapter.

1 Classical Systems and Concepts

Knowlton's movie [47] about list processing using the programming language L6 was one of the first experiments to visualize program behavior with the help of animation techniques. Other early efforts often focused on aiding teaching [1,44] including the classic "Sorting Out Sorting" [3,2] that described nine different sorting algorithms and illustrated their respective running times.

Experiences with algorithm animations made by hand and the wide distribution of personal computers with advanced graphical displays in the 1980's led to the development of algorithm animation systems. The well known system Balsa [19,8] introduced the concept of *Interesting Events* (IE's) and with it the binding of several views to the same state. In this approach the key points of the program are annotated with IE's by the visualizer (the person who specifies the visualization). When those IE's are reached during execution, an event, parameterized with information about the current program state, is sent to the views. The successor Balsa II [9] was extended with step and break points and a number of other features. In Zeus [11], Cat [14], and the later Java based system Jcat [18] the views were distributed on several workstations.

The system Tango [72] implemented the path-transition paradigm [73,77] that permitted smooth and concurrent animations of state transitions. In its successor Polka [79], these properties were revised to facilitate easier design of concurrent animation actions. As a front-end of Polka, an interactive animation interpreter called Samba [75] (including the later Java based Jsamba) was developed. Samba consisted of a number of parameterized ASCII commands that performed different animation actions. Thus, programs written in any programming language could be animated simply by having them output Samba commands at interesting event points.

The system Pavane [67] was noteworthy in exploring a declarative animation paradigm in which program states were mapped to visualization states. The animation designer simply created this mapping initially, then the program ran and the appropriate mappings were maintained.

A number of other noteworthy algorithm visualization systems and tools have been developed over the years. Some of the earlier efforts include systems in

Smalltalk [52,32], the Aladdin system [45,42], the Movie system [6], animations for algorithms textbooks [37,38], and the Gaigs system [56].

Recently, a number of new systems have been introduced. Many of these newer systems were presented at workshops and conferences, including the GI workshop SV'2000 [27], the First International Program Visualization Workshop 2000 [82] and the Dagstuhl Seminar on Software Visualization 2001 [28]. In the next four sections, we briefly describe a few of the newer systems, as well as noteworthy earlier systems, with respect to four important dimensions: specification technique, visualization technique, language paradigm, and domain-specificity.

2 Specification Technique

An important practical task in creating algorithm visualizations is to specify how the visualization is connected or applied to the algorithm. SV researchers have developed a number of approaches to this problem. In this section we will examine some of the different approaches and the systems that use the approaches. Note that some of the systems could be classified in several categories.

2.1 Event Driven

The *interesting event* approach was pioneered by Balsa [19,8] and has been used in many algorithm animation systems including its successor Zeus [11]. As mentioned above, the visualizer identifies key points in the program and annotates these with IE's. Whenever an IE is reached during execution, a parameterized event is dispatched to the views.

The event-based framework Ganimal [31] offers some new features like alternative interesting events, alternative code blocks, mixing of post-mortem and live/online algorithm animation, visualization control of loops and recursion, etc. Annotations are provided by the Ganila language, which are compiled into Java. The generated code allows the association of meta-information (settings) with each program point of the algorithm. Consequently a graphical user interface can be used to change these settings at runtime of the animation.

The Animal system [69,68] also utilizes an event-based approach and provides a number of advanced features for animation presentation including dynamic flexibility of animation mappings, reverse execution, internationalization of animation content, and flexible import and export capabilities.

2.2 State Driven

An alternative approach is to specify a mapping between program and visualization states, usually constructed by the visualizer before program execution. As discussed above, the Pavane system was an early adopter of this technique [66, 65]. The declarative approach is also utilized by some newer systems. Leonardo [23,24] is an integrated environment for developing, executing and animating C programs. Visualizations are specified by adding declarations written in Alpha, a declarative language, to the C program. LEONARDO also supports full reversible execution by using a specialized virtual machine that executes the compiled C program.

DAPHNIS is an algorithm animation system based on the use of data flow tracing. Some aspects of abstraction in the visualization are produced fully automatically, but to prepare an animation, it is necessary to supply an external configuration script that specifies the graphical representation and rules of translation for all the variables to be visualized. To achieve spatial or temporal suppression of unimportant information, a special kind of the Petri net formalism is applied to describe the process of algorithm execution. The DAPHNIS system as well as its theoretical model are discussed further in this chapter [36].

Demetrescu, Finocchi, and Stasko provide a direct comparison of both the *interesting event* and *state mapping* approaches in this chapter [25], identifying some scenarios where one might be preferable to the other.

2.3 Visual Programming

Another technique for specifying algorithm animations is the use of visual programming techniques. Visual programming (VP) seeks to make programs easier to specify by using a visual notation for the actual program commands and statements. As a whole, VP is considered to be distinct from SV [63], but such a graphical notation itself is a kind of statical code/data visualization.

One well-known project to embed animation capabilities into a visual programming language (VPL) is the declarative VPL FORMS/3 [20] in which animation is done by maintaining a network of one-way constraints. The developers of this language integrated an extension of the path-transition paradigm into their language, resulting in a unique approach to algorithm animation, e.g. a seamless integration of algorithm animation into the language and on-the-fly exploratory programming of an algorithm animation.

An area related to VP is programming by demonstration (PbD). In PbD, a person demonstrates an example or an operation of a task, and the PbD-system infers a program for this task. DANCE [71,76] is a PbD-interface to the TANGO system. After the user demonstrates an animation scenario in a direct manipulation style graphical editor, the DANCE system generates ASCII code that specifies the animation. This code is then used as input to TANGO.

2.4 Automatic Animation

Perhaps the simplest way to specify an animation, at least for the algorithm developer, is to have the animation automatically generated. Total, automatic creation of algorithm animations is extremely difficult however [10], so systems have provided differing levels of automation to specify algorithm animation. Because automated animation creation requires little or no effort on the programmer's part, this approach is very well-suited to debugging [54].

An early system in this area, UWPI [43], provided automatic animation by using a small expert system that chose the visualization for the data structures and operations of an algorithm by attempting to infer the abstract data types used in the program. The system could display abstractions of higher-level data structures, even though it did not truly "understand" them.

Jeliot is a family of program animation environments some of which support the semi-automated paradigm by allowing users to define the visual semantics of program structures or to select the most adequate ones. One system was exclusively developed for novice programmers. It supports fully automatic animation, and does not allow any customization of the animation. Ben-Ari, Myller, Sutinen, and Tarhio give an overview of the Jeliot family and discuss some empirical evaluations of some of the systems in this chapter [5].

Another technique that fits this category is the use of special pseudo-code languages in which programmers implement their code, and then the animation is automatically produced. Algorithma 99 [21] is such an example system.

3 Visualization Technique

One of the most important tasks in SV is the design of the graphical appearance of a visualization or animation. The display design must address a number of different issues, e.g., what information should be presented, how should this be done, should there be a focus on the important elements, and so on. Brown and Hershberger give a good overview of fundamental techniques on this topic [12]. In this introduction we discuss three aspects of visualizing algorithms that have received much attention lately: 3D Algorithm Animation, Auralization, and Web Deployment.

3.1 3D Algorithm Animation

There may be several reasons for integrating 3D graphics into an algorithm animation system. The third dimension can be used for capturing time (history), uniting multiple views, and displaying additional information [16]. Both the systems Polka [81] and Zeus [15] were extended with 3D graphics versions. Brown and Najork further integrated their earlier work on the platform-dependent Zeus3D into the Jcat system. With the resulting Java-based system, 3D animations could be run in any standard web browser [17,55]. The 3D animations were implemented using the object-oriented, scene-graph based graphics library Java3D (plugin). In the Gasp system, Tal and Dobkin explored 3D animations of computational geometry algorithms also [83]. They created a library of geometric data types including operations that were furnished with animation instructions.

3.2 Auralization

In SV, audio can be used to reinforce and replace visual cues, to convey patterns, to identify important events in a stream of data, and to signal exceptional conditions [13]. Recently the mapping of information to musical sound using parameters such as rhythm, harmony, melody or special leitmotifs has been studied.

Caitlin [85] is a preprocessor for Pascal which allows a user to specify an auralization for each type of program construct (e.g. a `FOR` statement) on the basis of a hierarchical leitmotif design. Caitlin does not allow auralization of data, however. Empirical studies [86] of this system show that novice programmers could interpret the musical auralizations, that musical knowledge had no significant effect on performance, and that, in certain circumstances, musical auralizations can be used to help locate bugs in programs.

The musical data sonification toolkit Muse [51] provides flexible data mappings to musical sounds. The data can come from any scientific source. It is written for the SGI platform and supports different mapping types of data to sound, like timbre, rhythm, tempo, volume, pitch and harmony.

A similiar system is Faust [88], a framework for algorithm understanding and sonification testing. It allows simple mappings of algorithm events to sound parameters and requires programmers to manually tag events in their algorithms. Furthermore, the interested programmer can easily change sound synthesis algorithms and add new features and attributes to these algorithms.

3.3 Web Deployment

With the growing use of the World Wide Web as a generic application and display platform, a number of recent algorithm animation systems have focused on delivery of animations over the Web. The JSamba and JCat systems mentioned earlier are two examples. Other systems presenting animations over the Web include JHave [57], the Sort Animator [26], Jeliot [41], and Jawaa [62].

4 Language Paradigm

Different language paradigms may need different abstractions and entities to be visualized, due to their unique styles of computation and methods for problem solving. The survey by Oudshoorn, Widjaja, and Ellershaw [59] analyses the visualization requirements for a variety of programming paradigms and gives a simple taxonomy of program visualization. In the following section we consider the most important language paradigms and illuminate some example systems.

4.1 Imperative Programming Languages

The imperative paradigm is based on the procedural approach to solve problems. From this point of view, the developer of an algorithm animation has to find abstractions of variables, data structures, procedures/functions and control structures. The Balsa system [9] exemplifies this paradigm.

4.2 Functional Programming Languages

The most significant abstractions for functional languages are functions and data structures. A textual browser to view the trace of the evaluation of a lazy functional language is discussed in [87]. The system facilitates navigating over a trace and it can be used as a debugging tool or as a pedagogical aid in understanding how lazy evaluation works.

The KIEL System [7] is an interactively controlled computer system for the execution of first-order functional programs written in a simple subset of Standard ML. In contrast to the prior discussed system, it offers ways to visualize the evaluation process.

A formal model of traversing graphical traces of lazy functional programs is introduced by Foubister [35]. This model provides the visual representation of graph reduction by template instantiation, and solves some problems in displaying the reduction, e.g., the large size of the graphs or their planarity.

4.3 Object-Oriented Programming Languages

The object-oriented paradigm has much in common with the imperative language paradigm. As a consequence, visualizations of object-oriented programs typically consider abstractions of objects, including inter-object communication, in addition to the above-mentioned abstractions for imperative languages.

One of the papers in the following chapter [58] deals with solutions for the endemic problem of aliasing within object-oriented programs, i.e., a particular object can be referred to by any number of other objects via its reference. This fact may be unknown to the algorithm animation system and can cause problems for animations. The paper discusses analysis of the program to determine the extent of aliasing as well as a visualization of ownership trees of objects in Java programs.

SCENE [48] automatically produces scenario diagrams (event trace diagrams) for existing object-oriented systems. This tool does not provide visualization of the message flow in an object-oriented program, but by means of an active text framework it allows the user to browse several kinds of associated documents, such as source code, class interfaces or diagrams, and call matrices.

4.4 Logical Programming Languages

In logic programs, the interesting abstractions are clauses and unification. One of the classic systems for visualizing logic programs is the Transparent Prolog Machine TPM [34]. It uses an AND/OR tree model of the search space and the execution of the logic program is shown as a DFS search. Another system for presenting logic programs was discussed by Senay and Lazzeri [70].

5 Domain Specific Animations

The search for adequate abstractions of the specific properties of a particular domain, e.g., realtime algorithms, can be a challenge when animating algorithms in such a focused domain. To find such abstractions, knowledge regarding the types of objects and operations that are dominant in the special domain is necessary. Often, general algorithm animation systems can be used to build domain-specific animations, but the effort can be extensive and much more involved than that required for a more narrowly-focused system.

Tal presents a conceptual model for developing algorithm animation systems for constraint domains in this chapter [84]. She illuminates the practical implementation of this model on the basis of a few example systems, e.g., the GASP system, mentioned earlier.

5.1 Computational Geometry

In computational geometry, the task of finding abstractions of the data can be relatively easy if the program data contains positional information. Hence, it can be displayed without complicated transformation.

A generic tool for the interactive visualization of geometric algorithms is GEOWIN discussed by Bäsken and Näher in this chapter [4]. It is a C++ data type which can be interfaced with algorithmic software libraries such as LEDA.

The EVEGA system [46] is a Java-based visualization environment for graph algorithms and offers a set of features to create and edit graphs, to display visualizations and to perform comparisons of different algorithms. Furthermore, it supports a relatively straightforward implementation of new algorithms by class extension.

5.2 Concurrent Programs

The animation of concurrent programs, which are typically very complex and large, must address a number of problems with regard to data collection, data display and program execution. Some problems encountered are inherently visual, e.g., a cycle in a resource allocation graph corresponds to a deadlock situation. Other problems can be a non-deterministic occurrence of a bug in program execution, which requires a clever visualization of the concurrent program. For an overview of systems for visualizing concurrent programs, see [49].

The event-based PARADE environment [74] supports the design and implementation of animations of parallel and distributed programs. Interesting events may be received via program calls, through pipes or read from a file, similar to the aforementioned SAMBA interpreter. A particular component of the system gathers the events for each processor or process and allows the user to manipulate the order of these events, e.g. chronologically or logically [50]. The POLKA animation system is used in PARADE to build the graphical views.

VADE [53] is a client-server based system for visualizing algorithms in a distributed environment. It provides libraries that facilitate the automatic generation of visualizations, and supports web page creation for the final animation. Furthermore, VADE offers synchronization methods that maintain consistency.

5.3 Real-Time Animation

Some domains, such as network protocols, need to represent exact timing relationships in the underlying program or algorithm. Polka-Rc [80] is an extension of Polka with features for real-time animation, i.e., animation actions of precise timings, initiations and durations. It also provides a flexible multiprocess mapping between program and visualization. More precisely in Polka-Rc the program and its animation run as separate processes and communicate via sockets.

The Jotsa system [64] is a Java package for performing interactive web-based algorithm animations (especially for network protocols). It supports exact time animation, multiple independent synchronized views, panning, zooming and linking of collections of objects. In addition it has facilities for animation of user-defined event-driven and timer-driven simulations of network protocols.

5.4 Computational Models

Another domain of interest to algorithm animators is the animation of computational models of formal languages, sets, relations, and functions. These models are typically for mathematical reasoning, not for programming of real hardware and real applications.

Jflap [40] is a tool for creating and simulating several kinds of automata, i.e. finite automata, pushdown automata and Turing machines, and for converting representations of languages from one form to another. Jflap is written in Java and has been used both in and outside of the classroom.

Diehl and Kerren [29] discuss how generation of visualizations of computational models and the visualization of the generation process itself increase exploration. Four approaches of increased exploration in formal language theory and compiler design are introduced and each approach is exemplified by an implemented system. As an example of such a system, the authors characterize GaniFa [30], a Java applet for the visual generation and animated simulation of finite automata.

5.5 Animation of Proofs

A relatively unexplored area is the visualization of proofs in theoretical computer science education. An example is Scapa [60,61] a system for the animation of structured calculational proofs. This system generates both an HTML document (with the help of a converter) and a Java file from a proof written in LaTeX. The visualizer has to extend and modify these files. The proof animation is finally created by using an extended version of the Lambada tool, which is a Java-based reimplementation of Samba.

6 Conclusion

Much progress has been made in the field of Algorithm Animation since the first films motivating the field, such as *Sorting Out Sorting*, were made. In this chapter introduction, we have highlighted a number of the landmark systems that have been developed in the area, plus we have surveyed some new developments. A brief introduction like this, however, is in no way a comprehensive overview of the field. We encourage the reader to use this introduction and the articles in this chapter as a starting point for exploring other research and creating new systems and techniques.

References

1. R. Baecker. Towards Animating Computer Programs: A First Progress Report. In *Proceedings of the Third NRC Man-Computer Communications Conference*, 1973.
2. R. Baecker. Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 24, pages 369–381. MIT Press, Cambridge, MA, 1998.
3. R. Baecker (with assistance of Dave Sherman). Sorting out Sorting. 30 minute color film (distributed by Morgan Kaufmann Pub.), 1981.
4. Matthias Bäsken and Stefan Näher. GeoWin A Generic Tool for Interactive Visualization of Geometric Algorithms. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
5. Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. Perspectives on Program Animation with Jeliot. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
6. J. L. Bentley and B. W. Kernighan. A System for Algorithm Animation. *Computing Systems*, 4(1), Winter 1991.
7. R. Berghammer. KIEL: A Program for Visualizations of the Evaluation of Functional Programs (in German). In *S. Diehl and A. Kerren, editors: Proceedings of the GI-Workshop "Software Visualization" SV2000*, May 2000.
8. M. H. Brown. *Algorithm Animation.* MIT Press, 1987.
9. M. H. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.
10. M. H. Brown. Perspectives on Algorithm Animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.
11. M. H. Brown. ZEUS: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.
12. M. H. Brown and J. Hershberger. Fundamental Techniques for Algorithm Animation Displays. In John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization.* MIT Press, 1998.
13. M. H. Brown and J. Hershberger. Program Auralization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 10, pages 137–143. MIT Press, 1998.
14. M. H. Brown and M. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, Boulder, CO, 1996.
15. M. H. Brown and M. A. Najork. Algorithm Animation Using 3D Interactive Graphics. In *Proceedings of the 1993 ACM Symposium on User Interface Software and Technology*, pages 93–100, Atlanta, GA, November 1993.

16. M. H. Brown and M. A. Najork. Algorithm Animation Using Interactive 3D Graphics. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 9, pages 119–135. MIT Press, 1998.
17. M. H. Brown and M. A. Najork. Three-Dimensional Web-Based Algorithm Animations. Technical Report 170, Compaq Systems Research Center, July 2001.
18. M. H. Brown and R. Raisamo. JCAT: Collaborative Active Textbooks Using Java. In *Proceedings of CompuGraphics'96*, Paris, France, 1996.
19. M. H. Brown and R. Sedgewick. A System for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984.
20. P. Carlson, M. Burnett, and J. Cadiz. A Seamless Integration of Algorithm Animation into a Declarative Visual Programming Language. In *Proceedings Advanced Visual Interfaces (AVI'96)*, 1996.
21. A.I. Concepcion, N. Leach, and A. Knight. Algorithma 99: An Experiment in Reusability and Component-Based Software Engineering. In *Proceedings of the 31st ACM Technical Symposium on Computer Science Education (SIGCSE 2000)*, pages 162–166, Austin, TX, February 2000.
22. K. C. Cox and G.-C. Roman. Abstraction in Algorithm Animation. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 18–24, Seattle, WA, September 1992.
23. C. Demetrescu and I. Finocchi. A Technique for Generating Graphical Abstractions of Program Data Structures. In *Proceedings of the 3rd International Conference on Visual Information Systems (Visual'99)*, LNCS 1614, pages 785–792, Amsterdam, 1999. Springer.
24. C. Demetrescu and I. Finocchi. Smooth Animation of Algorithms in a Declarative Framework. *Journal of Visual Languages and Computing*, 12(3):253–281, June 2001.
25. Camil Demetrescu, Irene Finocchi, and John Stasko. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
26. H. L. Dershem and P. Brummund. Tools for Web-based Sorting Animations. In *Proceedings of the 29th ACM Technical Symposium on Computer Science Education (SIGCSE 98)*, pages 222–226, Atlanta, GA, February 1998.
27. Eds.: S. Diehl and A. Kerren. Proceedings of the GI-Workshop "Software Visualization" SV2000 (in German). Technical Report A/01/2000, FR 6.2 - Informatik, University of Saarland, May 2000. `http://www.cs.uni-sb.de/tr/FB14`.
28. S. Diehl, editor. *Software Visualization*, volume 2269 of *LNCS State-of-the-art Survey*. Springer Verlag, 2002.
29. S. Diehl and A. Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001*, pages 60–64. ACM, 2001.
30. S. Diehl, A. Kerren, and T. Weller. Visual Exploration of Generation Algorithms for Finite Automata. In *Implementation and Application of Automata, LNCS 2088*, pages 327–328, 2001.
31. Stephan Diehl, Carsten Görg, and Andreas Kerren. Animating Algorithms Live and Post Mortem. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
32. R. A. Duisberg. Visual Programming of Program Visualizations. A Gestural Interface for Animating Algorithms. In *Proceedings of the 1987 IEEE Computer Society Workshop on Visual Languages*, pages 55–66, Linkoping, Sweden, August 1987.

33. P. Eades and K. Zhang, editors. *Software Visualization.* World Scientific Pub., Singapore, 1996.
34. M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4):1–66, 1988.
35. S. Foubister. *Graphical Application and Visualisation of Lazy Functional Computation.* PhD thesis, Department of Computer Science, University of York, 1995.
36. Jaroslaw Francik. Algorithm Animation Using Data Flow Tracing. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
37. P. A. Gloor. AACE - Algorithm Animation for Computer Science Education. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 25–31, Seattle, WA, September 1992.
38. P. A. Gloor. Animated Algorithms. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 27, pages 409–416. MIT Press, Cambridge, MA, 1998.
39. P. A. Gloor. User Interface Issues for Algorithm Animation. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 11, pages 145–152. MIT Press, Cambridge, MA, 1998.
40. E. Gramond and S. H. Rodger. Using JFLAP to Interact with Theorems in Automata Theory. *SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.
41. J. Haajanen et al. Animation of User Algorithms on the Web. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pages 360–367, Capri, Italy, September 1997.
42. E. Helttula, A. Hyrskykari, and K.-J. Räihä. Graphical Specification of Algorithm Animations with Aladdin. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 892–901, Kailua-Kona, HI, January 1989.
43. R. R. Henry, K. M. Whaley, and B. Forstall. The University of Washington Illustrating Compiler. *Sigplan Notices: SIGPLAN '90*, 25(6):223–233, June 1990.
44. F. Hopgood. Computer Animation Used as a Tool in Teaching Computer Science. In *Proceedings IFIP Congress*, 1974.
45. A. Hyrskykari and K.-J. Räihä. Animation of Algorithms Without Programming. In *Proceedings of the 1987 IEEE Computer Society Workshop on Visual Languages*, pages 40–54, Linkoping, Sweden, August 1987.
46. S. Khuri and K. Holzapfel. EVEGA: An Educational Visualization Environment for Graph Algorithms. In *Proceedings of the 6th Annual Conference on Innovaton and Technology in Computer Science Education, ITiCSE 2001.* ACM Press, 2001.
47. K. Knowlton. L6: Bell Telephone Laboratories Low-Level Linked List Language. 16-minute black-and-white film, 1966.
48. K. Koskimies and K. Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *Proceedings of the 18th IEEE International Conference on Software Engineering*, pages 366–375. IEEE Computer Society Press, 1996.
49. E. Kraemer. Visualizing Concurrent Programs. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 17, pages 237–256. MIT Press, Cambridge, MA, 1998.

50. E. Kraemer and J. T. Stasko. Toward Flexible Control of the Temporal Mapping from Concurrent Program Events to Animations. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94)*, pages 902–908, Cancun, Mexico, April 1994.
51. S. K. Lodha, J. Beahan, T. Heppe, A. Joseph, and B. Zane-Ulman. MUSE: A Musical Data Sonification Toolkit. In *Proceedings of International Conference on Auditory Display (ICAD)*, Palo Alto, CA, USA, 1997.
52. R. L. London and R. A. Duisberg. Animating Programs Using Smalltalk. *Computer*, 18(8):61–71, August 1985.
53. Y. Moses, Z. Polunsky, and A. Tal. Algorithm Visualization For Distributed Environments. In *Proceedings of the IEEE Symposium on Information Visualization 1998*, pages 71–78, 1998.
54. S. Mukherjea and J. T. Stasko. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, September 1994.
55. M. A. Najork. Web-Based Algorithm Animation. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 506–511, 2001.
56. T. L. Naps. Algorithm Visualization in Computer Science Laboratories. In *Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education*, pages 105–110, Washington, DC, February 1990.
57. T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVE – An Environment to Actively Engage Students in Web-Based Algorithm Visualizations. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 109–113, Austin, TX, March 2000.
58. James Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing and Ownership. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
59. M. Oudshoorn, H. Widjaja, and S. Ellershaw. Aspects and Taxonomy of Program Visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*. World Scientific Press, Singapore, 1996.
60. C. Pape. *Animation of Structured Proofs in Education at University Level (in German)*. PhD thesis, University of Karlsruhe, Germany, 1999.
61. C. Pape and P. H. Schmitt. Visualizations for Proof Presentation in Theoretical Computer Science Education. In Z. Halim, Th. Ottmann, and Z. Razak, editors, *Proceedings of International Conference on Computers in Education*, pages 229–236. AACE - Association for the Advancement of Computing in Education, 1997.
62. W. C. Pierson and S. H. Rodger. Web-based Animation of Data Structures using JAWAA. In *Proceedings of the 29th ACM Technical Symposium on Computer Science Education (SIGCSE 98)*, pages 257–260, Atlanta, GA, February 1998.
63. B. A. Price, R. Baecker, and I. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
64. S. Robbins. The JOTSA Animation Environment. In *Proceedings of the 31st Hawaii Int. Conference on Systems Science*, pages 655–664, 1998.
65. G.-C. Roman. Declarative Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 13, pages 173–186. MIT Press, Cambridge, MA, 1998.
66. G.-C. Roman and K. C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22(10):25–36, October 1989.
67. G.-C. Roman, K. C. Cox, Donald Wilcox, and Jerome Y. Plun. Pavane: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3(2):161–193, June 1992.

68. G. Rössling and B. Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 2002. to appear.
69. G. Rössling, M. Schuler, and B. Freisleben. The ANIMAL Algorithm Animation Tool. In *Proceedings of the ITiCSE 2000 Conference*, pages 37–40, Helsinki, Finland, 2000.
70. H. Senay and S. G. Lazzeri. Graphical Representation of Logic Programs and Their Behavior. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 25–31, Kobe, Japan, October 1991.
71. J. T. Stasko. Using Direct Manipulation to Build Algorithm Animations by Demonstration. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, New Orleans, LA, USA.
72. J. T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990.
73. J. T. Stasko. The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.
74. J. T. Stasko. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technical Report GIT-GVU-95-03, 1995.
75. J. T. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997.
76. J. T. Stasko. Building Software Visualizations through Direct Manipulation and Demonstration. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 14, pages 187–203. MIT Press, Cambridge, MA, 1998.
77. J. T. Stasko. Smooth Continuous Animation for Portraying Algorithms and Processes. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 8, pages 103–118. MIT Press, Cambridge, MA, 1998.
78. J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization*. MIT Press, 1998.
79. J. T. Stasko and E. Kraemer. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2), 1993.
80. J. T. Stasko and D. S. McCrickard. Real Clock Time Animation Support for Developing Software Visualisations. *Australian Computer Journal*, 27(4):118–128, 1995.
81. J. T. Stasko and J. F. Wehrli. Three-Dimensional Computation Visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 100–107, Bergen, Norway, August 1993.
82. E. Sutinen, editor. *Proceedings of the First Program Visualization Workshop 2000*, Porvoo, Finland, 2000. Department of Computer Sience, University of Joensuu, Finland.
83. A. Y. Tal and D. P. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), 1995.
84. Ayellet Tal. Algorithm Animation Systems for Constrained Domains. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001*.
85. P. Vickers and J. L. Alty. CAITLIN: A Musical Program Auralisation Tool to Assist Novice Programmers with Debugging. In *Proceedings of International Conference on Auditory Display (ICAD)*, Palo Alto, CA, USA, 1996.

86. P. Vickers and J. L. Alty. Musical Program Auralisation: Empirical Studies. In *Proceedings of International Conference on Auditory Display (ICAD)*, Atlanta, GA, USA, 2000.
87. R. Watson and E. Salzman. A Trace Browser for a Lazy Functional Language. In *Proceedings of the Twentieth Australian Computer Science Conference*, pages 356–363, 1997.
88. J. R. Weinstein and P. R. Cook. FAUST: A Framework for Algorithm Understanding and Sonification Testing. In *Proceedings of International Conference on Auditory Display (ICAD)*, Palo Alto, CA, USA, 1997.

Specifying Algorithm Visualizations: Interesting Events or State Mapping?[⋆]

Camil Demetrescu[1], Irene Finocchi[2], and John T. Stasko[3]

[1] Dipartimento di Informatica e Sistemistica,
Università di Roma “La Sapienza”,
Via Salaria 113, 00198 Roma, Italy.
demetres@dis.uniroma1.it,
http://www.dis.uniroma1.it/~demetres/

[2] Dipartimento di Scienze dell’Informazione,
Università di Roma “La Sapienza”,
Via Salaria 113, 00198 Roma, Italy.
finocchi@dsi.uniroma1.it,
http://www.dsi.uniroma1.it/~finocchi/

[3] College of Computing / GVU Center,
Georgia Institute of Technology
Atlanta, GA 30332-0280.
stasko@cc.gatech.edu,
http://www.cc.gatech.edu/~john.stasko/

Abstract.

Perhaps the most popular approach to animating algorithms consists of identifying *interesting events* in the implementation code, corresponding to relevant actions in the underlying algorithm, and turning them into graphical events by inserting calls to suitable visualization routines. Another natural approach conceives algorithm animation as a graphical interpretation of the state of the computation of a program, letting graphical objects in a visualization depend on a program’s variables. In this paper we provide the first direct comparison of these two approaches, identifying scenarios where one might be preferable to the other. The discussion is based on examples realized with the systems Polka and Leonardo.

1 Introduction

One of the main issues in algorithm animation is the specification of the graphical abstractions that illustrate computations. Two problems arise in this context:

[⋆] Work supported in part by the project “Algorithms for Large Data Sets: Science and Engineering” of the Italian Ministry of University and of Scientific and Technological Research (ex MURST 40%) and by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT).

modeling graphical scenes and animation transitions, and binding the attributes and the animated behavior of graphical objects to the underlying algorithmic code. The power of a specification method is mainly related to its flexibility, generality, and capability to customize visualizations. In this setting, a common approach is to use conventional textual programming languages as specification tools. In general, a visualization specification language can be different from the language used for implementing the algorithm to be visualized, though they often coincide. An important factor that determines the connection of the visualization code with the algorithm implementation is how animation events are triggered by the underlying computation. One approach, dubbed *event-driven*, consists of identifying *interesting events* in the implementation code, corresponding to relevant actions of the algorithm, and turning them into graphical events by inserting calls to suitable animation routines, usually written in an imperative or object-oriented style. Another natural approach, dubbed *data-driven*, is to specify a *mapping* of the computation state into graphical scenes, usually declaring attributes of graphical objects to depend on variables of the underlying program. In this case, animation events are triggered by variable modifications. For a comprehensive discussion of other specification methods used in algorithm visualization, we refer the interested reader to [4,12,13,15,21].

This article provides the first direct comparison of the interesting event and state mapping specification styles. The two approaches are reviewed in more detail in Section 2. Section 3 addresses the problem of specifying a basic algorithm visualization and provides two different solutions for the Bubblesort algorithm, one event-driven and one data-driven, realized in the systems Polka [20] and Leonardo [9]. Further advanced aspects of algorithm visualization specification are considered in Section 4: the discussion is based on refinements and extensions of the Bubblesort visualization code given in Section 3. Section 5 addresses concluding remarks.

2 Two Visualization Specification Techniques

In this section we briefly review the event-driven and the data-driven visualization specification methods, listing some systems that instantiate the two approaches, and in particular the systems Polka [20] and Leonardo [9].

2.1 Event-Driven Approach

A natural approach to animating algorithms consists of annotating the algorithmic code with calls to visualization routines. The first step consists of identifying the relevant actions performed by the algorithm that are interesting for visualization purposes. Such relevant actions are usually referred to as *interesting events*. For instance, in a sorting algorithm the swap of two items can be considered an interesting event. The second step is to associate each interesting event with a suitable animation scene. In the sorting example, if we depict the values to be ordered as a sequence of sticks of different heights, the animation of a swap event might be realized by exchanging the positions of the two sticks corresponding to the values being swapped. Animation scenes can be specified by setting up suitable visualization procedures that drive the graphic system according to the actual parameters generated by the particular event. Alternatively, these visualization procedures may simply log the events in a file for a *post-mortem* visualization. Calls to the visualization routines are usually obtained by annotating the original algorithmic code in the points where the interesting events take place. This can be done either by hand or by means of specialized editors.

The event-driven approach is very intuitive and virtually any conceivable visualization can be generated in this way. Besides being simple to implement, interesting events are not necessarily low-level operations (such as comparisons or memory assignments), but can be more abstract and complex operations designed by the programmer and strictly related to the algorithm being visualized (e.g., the swap in the previous example, as well as a rotate operation in the management of AVL trees). Major drawbacks are invasiveness (even if the code is not transformed, it is augmented) and code ignorance allowance: the person who is in charge of realizing the animation has to know the source code quite well in order to identify all the interesting points.

A limited list of well-known systems based on interesting events include Balsa [3], Zeus [5], Tango [18], XTango [19], Polka [20], CAT [6], ANIM [2].

Polka. In this paper we will consider examples of visualizations based on interesting events realized with Polka. Polka is a system for visualizing programs written in C++. The system has two main foci: allowing designers to create animations with smooth, continuous movements and simplifying the overall process of developing algorithm animations. To build an algorithm animation with Polka, the developer annotates the program source with *Algorithm Operations*. These are Polka's version of Interesting Events. The developer also creates *Animation Scenes* that are procedures which perform an animation chunk and are written using the Polka graphics library. Finally, the developer specifies a mapping between algorithm operations and animation scenes. The Polka system distribution includes full source code and numerous animation examples. Versions of Polka for both the X Window System and Microsoft Windows exist. Further information can be found at the URL `http://www.cc.gatech.edu/gvu/softviz`.

2.2 Data-Driven Approach

Data-driven systems rely on the assumption that observing how variables of a program change provides clues to the actions performed by the underlying algorithm. The focus is on capturing and monitoring the data modifications rather than on processing the interesting events issued by the annotated algorithmic code. Specifically, data-driven systems realize a graphical mapping of the state of the computation (*state mapping*): an example is given by conventional debuggers, which provide a direct feedback of how variables change over time.

Specifying an animation in a data-driven system consists of providing a graphical interpretation of the *interesting data structures* of the algorithmic code.

It is up to the system to ensure that the graphical interpretation reflects at any time the state of the computation of the program being animated. In the case of conventional debuggers, the interpretation is fixed and cannot be changed by the user: typically, a direct representation of the content of variables is provided. The debugger just updates the display after each change, sometimes highlighting the latest variable that has been modified by the program to help the user maintain context. In a more general scenario, an adjacency matrix used in the code may be visualized as a graph with vertices and edges, an array of numbers as a sequence of sticks of different heights, and a heap vector as a balanced tree. As the focus is only on data structures, the same graphical interpretation, and thus the same visualization code, may be reused for any algorithm that uses a given data structure. For instance, any sorting algorithm that manages to reorganize a given array of numbers may be animated with exactly the same visualization code that displays the array as a sequence of sticks. Main advantages of the data-driven approach are a clean animation design and a high ignorance of the code: in most cases only the interpretation of "interesting variables" has to be known in order to produce a basic animation. On the other hand, focusing only on data modification may sometimes limit customization possibilities, making it difficult to realize animations that would be natural to express with interesting events. As we will see in Section 4, a *pure state mapping* approach, where there is no connection of the visualization code with the program's control flow, is intrinsically less powerful than interesting events.

Examples of systems based on state mapping are Pavane [14,16], Leonardo [9], and WAVE [10]. Toolkits such as CATAI [8], Gato [17] and LEDA [11] provide self-animating data structures, incorporating the principles of state mapping, but still supporting interesting events. Declarative visual programming languages that integrate algorithm animation capabilities have been also considered (see, for instance, Forms/3 [7]).

Leonardo. In this paper we will consider examples of state mapping visualizations realized with Leonardo. Leonardo is an integrated environment for developing, executing, and visualizing C programs. It provides two major improvements over a traditional integrated development environment. In particular, it supports a mechanism for visualizing computations graphically as they happen by attaching in a declarative style graphical representations to key variables in a program. With this technique, basic animations can usually be obtained with a few lines of additional code. It is to notice that Leonardo does not realize a pure state mapping, in the sense that it allows the user to control in an imperative style which visualization declarations are active at any time. However, differently from the interesting events, these manipulations change the set of active declarations, rather than the visualization itself, and may not have necessarily an immediate effect on the graphical scene. As a second main feature, Leonardo includes the first run-time environment that supports fully reversible execution of C programs. The system is distributed with a collection of animations of more than 60 algorithms and data structures including approximation, combinatorial optimization, computational geometry, on-line, and dynamic algorithms. Leonardo has been widely distributed on CD-ROM in computer magazines and is available for download in many software archives over the Web. It has received several technical reviews and more than 18,000 downloads during the last two years. At the time of writing, Leonardo is available only on the Macintosh platform. Further information can be found at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

3 Anatomy of a Basic Visualization Specification

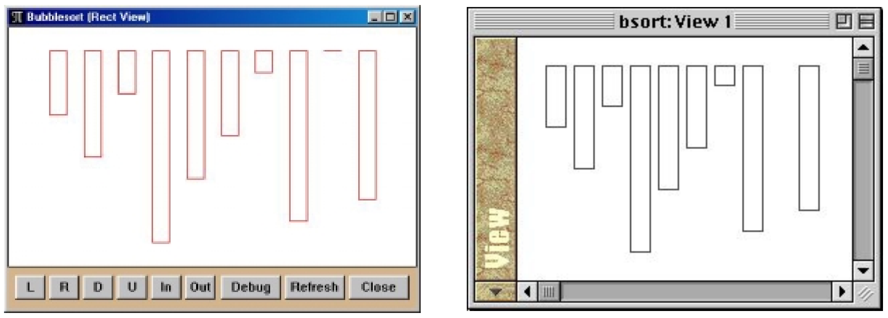
In this section we show how to specify a simple algorithm visualization using interesting events and state mapping. In particular, we focus on sorting algorithms and we show how to specify the well-known *sticks visualization*, where items to be sorted, assumed to be non-negative numbers, are visualized as rectangles of height proportional to their values. We first describe how the final visualization should look, and then we provide two solutions for the Bubblesort algorithm: one event-driven, realized with Polka, and one data-driven, realized with Leonardo. We give and discuss actual code and screenshots from both. For simplicity, we do not address issues of interaction with the visualization.

Bubblesort Code. We base our visualization examples on the following C/C++ implementation of the Bubblesort algorithm, which sorts an array `v` of `n` integer values.

```
1.   int v[]={3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
2.   void main(void) {
3.       for (j=n; j>0; j--)
4.           for (i=1; i<j; i++)
5.               if (v[i-1]>v[i]) {
6.                   int temp=v[i]; v[i]=v[i-1]; v[i-1]=temp;
7.               }
8.   }
```

In this implementation, the first pass of lines 4–7 scans the first n elements, the second pass scans the first $n-1$ elements, etc. As elements are being swapped, each pass leaves the highest element found at its final proper position.

Visualization Setup. The first steps in specifying an algorithm visualization consist of deciding which pieces of information related to the algorithm's execution should be visualized and choosing a suitable graphical representation for them. In the case of sorting algorithms, an effective visual methaphor is to associate sticks of different heights to elements to be sorted. A possible simple layout places sticks vertically from left to right aligning their tops at the top of the viewport (see Figure 1). A swap operation can be animated in many ways: perhaps the simplest one is to show consecutive scenes that visualize the sticks before and after the swap.



(a) (b)

Fig. 1. Screenshots of the Bubblesort visualization in: (a) Polka; (b) Leonardo.

Polka. Visualizations are specified in Polka by annotating the program source with interesting events. Below, we show the source code for the Bubblesort program that has been annotated with interesting event calls.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--)
        for (i=1; i<j; i++)
            if (v[i] > v[i+1]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
}
```

Two events exist here. The first, "`Input`", signifies that all the array values to be sorted are set and that the animation should draw the initial configuration of the array. The event must send the size of the array and the array itself to the animation component as parameters.

We will omit the animation scene that is invoked as a response to the "`Input`" event for brevity. This scene creates and lays out the set of vertical rectangles and stores them in an array of Polka `Rectangle` objects, which is a subclass of the basic graphic primitive `AnimObject`. The scene does involve some subtle geometric calculations, however, as the designer must position all the rectangles with their tops aligned, space the rectangles out horizontally, and scale the heights of the rectangles according to the corresponding array values. Frequently, this type of geometric layout is the most difficult aspect of creating an algorithm animation.

The second event, "`Exchange`", signifies that a swap of two elements has occurred. It passes the indices of the two exchanged array elements as parameters. The corresponding animation code for this event is shown below.

```
int Rects::Exchange(int i, int j) {
 2.     Loc *loc1 = blocks[i]->Where(PART_NW);
 3.     Loc *loc2 = blocks[j]->Where(PART_NW);
 4.     Action a("MOVE",loc1,loc2,1);
 5.     Action *b = a.Reverse();
 6.     int len = blocks[i]->Program(time,&a);
 7.     time = Animate(time,len);
 8.     len  = blocks[j]->Program(time,b);
 9.     time = Animate(time,len);
10.     Rectangle *t = blocks[i]; blocks[i] = blocks[j]; blocks[j] = t;
11.     return len;
12.  }
```

First, we get the top-left (`NW`) corners of the two appropriate rectangles (lines 2–3), and then we create two movement `Action`s between them, in the two opposite directions (lines 4–5). Next, we schedule the first block's animation to occur at the current time; animate it; schedule the second block's animation; and animate it. Finally, we must swap the two objects being held int the `Rectangle` `AnimObject` array. This animation routine makes the first rectangle move in one sudden jump, then the second rectangle moves afterward, again in one jump. Note that the variables `blocks` and `time` are defined in this particular View of the animation which is a C++ class of type `Rects` here. A screenshot of the resulting visualization in Polka is shown in Figure 1a.

Leonardo. Visualizations are specified in Leonardo by adding to C programs declarations written in ALPHA, a simple declarative language, enclosing them with separators `/**` and `**/`. A complete ALPHA specification of the sticks visualization described above is shown below; this fragment can be simply appended to the Bubblesort code and compiled in Leonardo.

```
 1.  /**
 2.      View(Out 1);
 3.      Rectangle(Out ID,Out X,Out Y,Out L,Out H,1)
 4.          For N:InRange(N,0,n-1)
 5.          Assign X=20+20*N  Y=20  L=15  H=15*v[N]  ID=N;
 6.  **/
```

In line 2 we declare a window with identification number 1: this window is the container of the visualization. Sticks are then declared in lines 3–5, where we enumerate `n` rectangles (line 4), and we locate them in the local coordinate system according to the desired layout (line 5). Specified geometrical attributes include the coordinates of the left-top corner (`X`,`Y`), the width `W`, and the height `H` of the `N`-th rectangle, with `N` $\in [0, n-1]$. The last parameter in the `Rectangle` declaration (line 3) makes sticks appear in window 1. Like windows, rectangles have identification numbers (`ID`): this allows us to refer to them in any subsequent declarations.

Observe that the ALPHA code refers to variables `v` and `n` of the underlying C program, and size and position of sticks depend on them: Leonardo reevaluates automatically the ALPHA code and updates the graphical scene for each change of these variables. Since both statements `v[i]=v[i-1]` and `v[i-1]=temp` in line 6 change the array `v`, this yields two animation events per swap. A screenshot of the resulting visualization in Leonardo is shown in Figure 1b.

Even if imperative state mapping specification has also been considered (see WAVE [10]), the declarative approach has many advantages: in particular, the programmer is encouraged to think in terms of "what she wants", and not in terms of "how to obtain it" (see, e.g., Section 4.3). A price paid for this, however, may be a steeper learning curve for programmers who have never used a declarative language.

4 Customizing Visualizations

The task of specifying a visualization usually proceeds incrementally through different levels of sophistication. In Section 3 we have shown how to specify the well-known sticks visualization of the Bubblesort algorithm. As the power of a specification method is mainly related to flexibility, generality, and capability of customizing visualizations, we now consider some refinements of the basic Bubblesort animation, discussing further aspects of interesting events and state mapping.

4.1 Specifying the Granularity of Animations

We use the word *granularity* to indicate the level of detail of animation events: for instance, a sorting animation where items being swapped are moved one at a time, as in the example in Section 3, is characterized by a higher granularity (closer to the actual code that uses a temporary variable for the swap) than the one where both items are moved simultaneously (elementary steps are logically grouped and details elided).

There is a main difference in the way granularity is controlled with interesting events and state mapping. To generate an animation event with interesting events, a function has to be called: thus, increasing the number of animation events requires increasing the number of function calls, so the granularity is low by default. With state mapping, each change of a variable being mapped into some graphical object yields automatically an animation event, so the granularity is high by default: to control granularity we therefore need a mechanism to prevent variable changes from being automatically turned into animation events. In the following, we show how to modify the Bubblesort visualization code presented in Section 3 in order to reduce the granularity in the swap animation.

Polka. Making the two blocks exchange positions simultaneously, rather than sequentially, is straighforward in Polka. We simply schedule their movement Actions to commence at the same animation time, and then we animate after that. The code below, when substituted into the `Exchange` animation routine of Section 3, performs this concurrent animation.

```
    int len = blocks[i]->Program(time,&a);
    blocks[j]->Program(time,b);
    time = Animate(time,len);

```

Notice that we have reduced the number of visualization instructions in order to reduce the number of animation events.

Leonardo. Leonardo provides a simple mechanism for controlling the granularity: if the predicate `ScreenUpdateOn` is declared, then variable changes trigger automatically updates of the visualization. If `ScreenUpdateOn` is not declared, then the visualization system is idle and no animation events occur. To let each swap in our example produce just one animation event, we can temporary suspend screen updates while the swap occurs: we just "undeclare" `ScreenUpdateOn` before the swap, and redeclare it thereafter, as shown below.

```
/** Not ScreenUpdateOn; **/
int temp=v[i]; v[i]=v[i-1]; v[i-1]=temp;
/** ScreenUpdateOn; **/
```

Notice that we have increased the number of the visualization instructions in order to reduce the number of animation events.

4.2 Accessing vs. Modifying Data Structures

Sometimes we might be interested in visualizing actions of an algorithm corresponding to no variable modification: consider, for instance, events of comparison of two elements in a sorting algorithm to decide whether they need to be swapped. It is easy to animate such actions with interesting events, which can be associated to any conceivable algorithmic event. On the contrary, this seems to be a major problem with state mapping, where animation events can result only from variable changes.

Polka. Suppose that we wish to illustrate the comparison of two array elements to determine whether they need to be exchanged. To do so, we add a new interesting event named "`Compare`" to the Bubblesort source code. This event occurs just before the actual value comparison is made in the program, and it passes the two pertinent array indices as parameters. Below, we show the modified program source.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--)
        for (i=1; i<j; i++) {
            bsort.SendAlgoEvt("Compare",i,i-1);
            if (v[i-1] > v[i]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
        }
}
```

Let us suppose that we want to illustrate the comparison operation in the program by flashing the two corresponding rectangles in the animation. This is performed in Polka by modifying the fill value of the `Rectangle AnimObjects`. Originally, the rectangles have a fill value of 0.0, indicating that they are simply outlines (1.0 signifies solid color fill, and values in between correspond to intermediate fills). We create a "`FILL`" animation action of two frames that takes the rectangle from empty, to half-filled, and back to empty. We then make a new `Action` that is this simple fill change iterated four times, thus making the flashing effect more striking. We schedule this behavior into both rectangles and perform the animation.

```
int Rects::Compare(int i, int j) {
    double flash[2];
    flash[0] = 0.5;
    flash[1] = -0.5;
    Action a("FILL",2,flash,flash);
    ActionPtr b = a.Iterate(4);
    int len = blocks[i]->Program(time,b);
    len = blocks[j]->Program(time,b);
    time = Animate(time, len);
    return len;
}
```

Leonardo. A general way to illustrate a comparison event in Leonardo is to "simulate" an interesting event. To do so, we add to the Bubblesort program a new C function, `void Compare(int i,int j)`, which takes the two pertinent array indices as parameters. This function is invoked just before the comparison and highlights the elements being compared, very much like the interesting event "`Compare`" does in the Polka code.

```
1.  void Compare(int i, int j) {
2.     int k=0;
3.     /** RectangleColor(ID,Out Grey,1) If (ID==i || ID==j) && k%2; **/
4.     while (k<8) k++;
5.  }
```

The main idea is to create a dummy sequence of variable changes: to this aim, we declare a local variable `k`, whose value flips four times from even to odd and back to even (line 4). The declaration in line 3, whose scope is local to the function body, just states that rectangles with `ID`s equal to `i` or to `j`, corresponding to the elements being compared, must have solid gray color fill whenever `k` is odd (i.e., `k%2` is non-zero). Even if this unusual method for defining animations may seem strange at first sight, mixing declarative and imperative specification yields great flexibility in the animation design.

Notice that variable i in the Bubblesort code indicates which items are currently being compared: thus, highlighting them would be easy in a pure declarative style. However, this would be an indirect way of portraying comparison events and might not be applicable to other algorithms.

4.3 Visualizing Invariant Properties of Algorithms

An important issue in algorithm visualization is portraying invariant properties of a program, which usually provide a sound foundation to the algorithm's correctness or performances. Visualizing invariant properties can help discover implementation errors and foster a better comprehension of combinatorial, algebraic, or numerical aspects of the problem at hand.

An interesting invariant property of the Bubblesort code shown in Section 3 is that array elements with indices greater than or equal to `j` are always at their final proper positions. We note that, since eventually `j` gets equal to zero, this implies the correctness of the whole procedure. To highlight sticks corresponding to elements that are properly positioned "in place", we color them red.

Polka. We create a new interesting event titled "`InPlace`" taking one parameter, the index of the array value now at its final position. We insert this event at the end of the outer of the two main loops in the code. We also must add one final event at the very end of the algorithm.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--) {
        for (i=1; i<j; i++) {
            bsort.SendAlgoEvt("Compare",i,i-1);
            if (v[i-1] > v[i]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
        }
        bsort.SendAlgoEvt("InPlace",j-1);
    }
    bsort.SendAlgoEvt("InPlace",0);
}
```

To indicate that an array element is in place, we change it from a simple outline to a solid colored rectangle and we change its color to red. This animation routine uses the "`FILL`" `Action` much as the `Compare` animation routine did as well as a simple color change `Action`. We schedule both to occur at the same time, and a one frame animation results.

```
int Rects::InPlace(int i) {
    double f = 1.0;
    Action a("FILL",1,&f,&f);
```

```
        Action b("COLOR","red");
        blocks[i]->Program(time,&a);
        int len = labels[i]->Program(time,&b);
        time = Animate(time, len);
        return len;
   }
```

Leonardo. To achieve the same result in Leonardo, we just need to add the following declaration to the ALPHA code given in Section 3.

```
        RectangleColor(ID,Out Red,1) If ID>=j;
```

Notice that the declarative specification allows us to encode directly the invariant property described above: here we state that rectangles in window 1 having `ID` greater than or equal to `j`, which correspond to array elements with indices greater than or equal to `j`, should have red color.

4.4 Adding Smooth Animation

Smooth animation is a very useful addition to algorithm visualizations for creating continuity in the display and for capturing the user's attention. In this section we address the problem of specifying smooth animations using interesting events and state mapping. In particular, we modify the Bubblesort example to visualize swaps as smooth transitions along curved paths, rather than jerky movements. While obtaining the desired solution with interesting events is straightforward in Polka, the Leonardo implementation involves some subtle considerations.

Polka. Changing the exchange operation's movement animations from being one frame "jumps" to smooth, multiframe, curved motions is very easy with Polka. We simply change the one line of the `Exchange` animation routine that constructs the movement path. We use a different `Action` constructor, one that utilizes the predefined `CLOCKWISE` trajectory taking 20 animation frames.

```
      Action a("MOVE",loc1,loc2,CLOCKWISE);
```

Leonardo. If the predicate `SmoothAnimationOn` is declared, Leonardo provides automatic in-betweening of graphical scenes: changes of graphical objects by the same IDs in consecutive scenes are automatically linearly interpolated to generate intermediate frames. In order for this to work properly, no two graphical objects can have the same ID.

We now consider the animation effect obtained by simply adding to our Bubblesort visualization code the following declaration.

```
      SmoothAnimationOn;
```

Since a rectangle with `ID=x` has height proportional to `v[x]` in our implementation (see line 5 of the visualization code in Section 3), swaps are seen from the viewpoint of array slots, which get their content changed. Thus, the swap animation resulting from declaring `SmoothAnimationOn` is that one stick grows and one shrinks. This might be fine anyway, but we were expecting sticks to jump, not change in size as in an animated histogram.

To customize the behavior to the desired result, we can look at swaps from the viewpoint of elements that move from slot to slot: to do so, we just let `ID=v[N]` instead of `ID=N` in line 5 of the visualization specification. In this way, rectangles by the same ID may have different positions before and after a swap, but same size. To meet the requirement that no two graphical objects can have the same ID, now we have the constraint that the array must contain no duplicates. We might also need to revise previous declarations that assumed `ID=N`, e.g., predicate `RectangleColor` in Section 4.3. At this point, to move sticks along curves instead of straight lines, we can further customize the animation, adding the following declaration.

```
8.     RectanglePosPath(ID,Out Curve,1);
```

Here we declare that any change of position of a rectangle in window 1, regardless of its ID, must be interpolated along a curve. Notice that the animation behavior of a graphical object is specified in Leonardo as an attribute of the object itself.

4.5 Other Issues

Showing Computation History. While invariant properties of programs are easily visualized with state mapping since they are defined on the current computation state, visualizing the history of the computation may be difficult, unless the current state of the computation includes some information about previous states. In fact, the solution usually adopted with both interesting events and state mapping is to record some history of previous states in a data structure, which is accessed for generating the visualization. Another possible solution with state mapping is to record the history in the mapping itself, which is progressively enriched with new declarations as the program runs.

Animating Multi-phase Algorithms. Some algorithms are based on several internal phases, each of which should be visualized in a different way (see, e.g., Ford-Fulkerson's maxflow algorithm [1]). This is achieved quite naturally with interesting events. Visualizing multi-phase algorithms with a pure state mapping, instead, may be difficult: the problem is easily solved, however, if the system allows us to let the graphical interpretation of variables depend upon the portion of code that is currently being executed. Leonardo, for instance, provides ad-hoc directives (`Not x`, `Assert x`, `Negate x`, `Substitute x With y`) that can activate, deactivate, or replace previous declarations at any point of the algorithmic code. We remark that, even if this is still state mapping, it is not "pure" in the sense that the set of active declarations defining the mapping depends on the program's control flow and is manipulated in an imperative style.

5 Conclusions

In this paper we have addressed specification aspects in algorithm visualization, providing the first direct comparison of the two most commonly used specification methods: interesting events and state mapping. We have based our discussion on specifying the well-known sticks visualization of the Bubblesort algorithm in the systems Polka and Leonardo, which instantiate the two approaches.

While interesting events are very intuitive and well-suited for specifying highly customized animations, they usually require developers to write several lines of additional code even for basic animations, and may lack in code ignorance allowance. On the other side, specifying visualizations with state mapping usually requires developers to write few lines of additional code, and little knowledge of the underlying code is needed, but this method may have a steeper learning curve and appears to be less flexible than interesting events in some customization aspects. It is our opinion that devising systems able to support both the declarative and the imperative visualization specification styles would represent an interesting research contribution, likely to be best suited for deployment in concrete applications.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.
2. J. Bentley and B. Kernighan. A System for Algorithm Animation: Tutorial and User Manual. *Computing Systems*, 4(1):5–30, 1991.
3. M.H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.
4. M.H. Brown. Perspectives on Algorithm Animation. In *Proceedings of the ACM SIGCHI'88 Conference on Human Factors in Computing Systems*, pages 33–38, 1988.
5. M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 7-th IEEE Workshop on Visual Languages*, pages 4–9, 1991.
6. M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 12th IEEE International Symposium on Visual Languages (VL'96)*, pages 266–275, 1996.
7. P. Carlson, M. Burnett, and J. Cadiz. Integration of Algorithm Animation into a Visual Programming Language. In *Proc. Int. Workshop on Advanced Visual Interfaces*, 1996.
8. G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998. To appear in Journal of Visual Languages and Computing. System home page: `http://isis.dia.unisa.it/catai/`.
9. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

10. C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing Algorithms over the Web with the Publication-driven Approach. In *Proc. of the 4-th Workshop on Algorithm Engineering (WAE'00)*, LNCS 1982, pages 147–158, 2000.
11. K. Mehlhorn and S. Naher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambrige University Press, ISBN 0-521-56329-1, 1999.
12. B.A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
13. B.A. Price, R.M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
14. G.C. Roman and K.C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22:25–36, 1989.
15. G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26:11–24, 1993.
16. G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
17. A. Schliep and W. Hochstättler. Developing Gato and CATBox with Python: Teaching Graph Algorithms through Visualization and Experimentation. In *Proceedings of Multimedia Tools for Communicating Mathematics (MTCM'00)*, 2000.
18. J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.
19. J.T. Stasko. Animating Algorithms with X-TANGO. *SIGACT News*, 23(2):67–71, 1992.
20. J.T. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.
21. J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.

Perspectives on Program Animation with Jeliot[*]

Mordechai Ben-Ari[1], Niko Myller[2], Erkki Sutinen[2], and Jorma Tarhio[3]

[1] Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100, Israel
moti.ben-ari@weizmann.ac.il

[2] Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
{nmyller,sutinen}@cs.joensuu.fi

[3] Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
jorma.tarhio@hut.fi

Abstract.

The Jeliot family consists of three program animation environments which are based on a self-animation paradigm. A student can visualize her Java code without inserting additional calls to animation primitives. The design of the animation environments has been guided by the analysis of feedback from high school and university students. Evaluation studies indicate the benefit of dedicated animation environments for different user groups like novice programmers. Based on the results of these studies, we present plans for a future work on Jeliot.

1 Introduction

The Jeliot family consists of three program animation environments: Eliot [1], Jeliot I [2], and Jeliot 2000 [3], which have been developed in order to improve the teaching and learning of computer science, in particular, programming. The key design principle has been learning-by-doing: a student should have an animation tool which helps him to easily construct a visual representation of a program. The two representations of a program, namely its code and its animation, should match the mental image of the student, so that he could concentrate on the comprehension process instead of being misled by disturbing visual clues.

To achieve the kind of transparency in an animation environment described above, you need a consistent technical design. The Jeliot framework is based upon *self-animation*: the syntactical structures of a programming language have

[*] The work was supported by the National Technology Agency, Finland.

built-in visual semantics. In this way, an interpreter or compiler can automatically generate the animation of a program. This means that a student can, in principle, write any program without worrying about how to visualize it. The learning process takes place at the level of coding a program and simultaneously studying the two representations, textual and visual, not by creating the textual representation, and then subsequently creating another—visual—one for existing code.

A danger of using an automated system in education is that the student may exhibit superficial learning, without having made the personal effort to truly understand to subject. Or worse: does a system make a student into a zombie (as a graduate student from Ekaterinburg commented), by forcing him to see the running-time behavior of a program in a predetermined way? These kinds of questions led us to create a semi-automatic implementation of the self-animation paradigm. A semi-automatic visualization environment should allow a student to define the visual semantics for each of the program structures, or at least to choose the most appropriate one for his needs.

Apparently, there is a trade-off between the speed at which a fully-automatic animation can be constructed, and the versatility of the semi-automatic paradigm. One member of the Jeliot family is fully-automatic, rather others retain the flexibility of the semi-automatic paradigm. Thus, the Jeliot framework offers an attractive platform for evaluating the effects of various animation strategies in different student populations.

2 The Development of the Jeliot Family

Jeliot is an animation tool for visualizing the execution of Java programs or algorithms. We review two versions of Jeliot: Jeliot I [2,4] implemented in the University of Helsinki works on the Web, and Jeliot 2000 [3] implemented in the Weizmann Institute of Science is a single Java application. We also consider Eliot [1], the predecessor of Jeliot I, as a member of the Jeliot family, because Eliot is functionally similar to Jeliot I. The name Eliot was taken from the Finnish word *Eliöt*, which means living organisms. The name Jeliot stands for Java-Eliot.

2.1 Early Years

The Jeliot family is an outcome of a long process. In 1992 Erkki Sutinen and Jorma Tarhio were involved with a project [5] of implementing ready-made animations for string algorithms. They noticed that the actual process of creating animations was more useful for learning than just watching ready-made animations. Because it took up to 100 hours to create a simple animation using the tools that were available at that time, the development of new tools for creating animations was started.

The first step towards Eliot was the implementation of self-animating data types. A data type is *self-animating* if the animation system provides a selection of visual representations for the type, and predefined animations are associated with its operations. If a program uses animated data types, its animation is seen as a sequence of the visualized operations when the program is run. This paradigm is called *self-animation*.

Self-animation is similar to the interesting-events approach [6], where the events are connected with the operations of the data types. However, self-animation has several advantages over the interesting-events approach: The algorithm and animation codes are not separated and data is not duplicated, because you do not have to construct the animation by inserting calls to animation primitives within the code of the algorithm. With self-animation, the preparation of an animation for a new algorithm is fast, and code reuse is easier.

Related Systems. The animation of Jeliot is controlled by operations on data structures. This kind of animation is closely connected with the development of debuggers and has a long history. Incense [7] was probably the first system capable of showing data structures of several kinds. Provide [8] offers alternative visual representations for variables. PASTIS [9] is an example of associating animation with a debugger. UWPI [10] introduced sophisticated automatic animation. In UWPI, a small expert system selects the visualization for a variable based on naming conventions of data types. At least Lens [11], VCC [12], and AAPT [13] are worth mentioning among other animation systems related to Jeliot.

2.2 Eliot

The key features of Eliot are self-animating data types and a user interface. Eliot extracts and displays the names of the variables of the self-animating types, which are integer, real, character, array and tree. The user decides which variables should be animated and selects their visual appearance. The user may accept the default values or change them individually for each object. In this way, constructing an animation is semi-automatic: the basic operations are defined automatically by the code, while the user can fine-tune the animation in the second phase, according to his internal view of data structures.

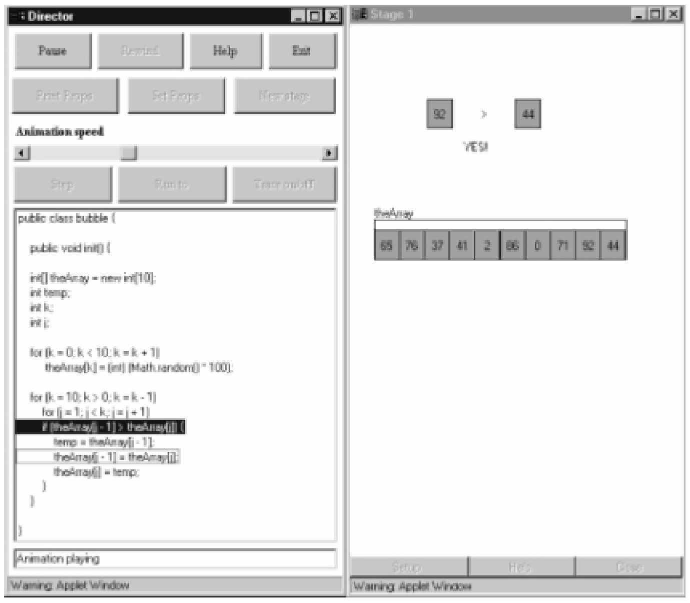
It is the integrated user interface of Eliot which makes self-animation practical to use. With Eliot it takes only a few minutes to design and compile a simple animation for a C program, but the same process would take about an hour without the user interface, because the set of animated variables must be programmed and the values of all visual attributes set by hand.

Eliot supports multiple animation windows called *stages*, which can be displayed simultaneously. The selection of the animated variables and their characteristics on each stage is independent. Eliot provides a feature to store the selection of variables and their visual parameters for later use.

Presentation of animation in Eliot is based on a *theater metaphor* [1], which has guided the design as well as the implementation. One can see the entire animation as a theatrical performance. The script of a play involves a number of roles, where the roles correspond to the variables of the algorithm to be visualized. An actor plays a role: in an animation, an actor is a visual interpretation of a variable. A play may have many simultaneous directions on multiple stages; similarly, an algorithm might have different visualizations on multiple animation windows.

2.3 Jeliot I

Eliot was completed in 1996. It ran under X windows and used the Polka animation library [14]. Because porting Eliot would have have been difficult, we decided to create a similar system for the World-Wide Web that would be portable. The Jeliot I system for animating Java programs was finished in 1997. Both Eliot and Jeliot I were implemented by students of University of Helsinki under direction of Erkki Sutinen and Jorma Tarhio.



```
public class bubble {

   public void init() {

   int[] theArray = new int[10];
   int temp;
   int k;
   int j;

   for (k = 0; k < 10; k = k + 1)
       theArray[k] = (int) (Math.random() * 100);

   for (k = 10; k > 0; k = k - 1)
      for (j = 1; j < k; j = j + 1)
      if (theArray[j - 1] > theArray[j]) {
         temp = theArray[j - 1];
         theArray[j - 1] = theArray[j];
         theArray[j] = temp;
      }
   }

}
```

Animation playing

Warning: Applet Window

Warning: Applet Window

Fig. 1. A screenshot of Jeliot I

Although the functionality of Jeliot I is similar to that of Eliot, the technical design is completely different and is based on client-server architecture. Moreover, the Polka library is not any more used; instead, graphical primitives were implemented in using the standard Java libraries. The implementation of self-animation in Eliot relied on the ability to overload operators in the underlying implementation language, C++. In Jeliot I, calls of relevant animation primitives are inserted into the algorithm during preprocessing of the source code.

Jeliot I is capable of animating all the primitive types of Java (boolean, integral, and floating-point types), one- and two-dimensional arrays, stacks and queues. For all types, visual representations can be selected. However, the animated tree type of Eliot is not supported. Jeliot I highlights the active line of code in the program window during execution.

In the terms of the theater metaphor, Jeliot I has an additional feature that did not exist in Eliot. Jeliot I enables *improvisations*, where the user can modify the visual appearance on the stage while a performance is running. The modified visualization parameters have an immediate impact on the actors on stage.

Figure 1 shows a screen capture from Jeliot I. The main control panel is on the left; on the right is one stage upon which an animation is taking place. There are many control windows used to configure the animation, too many in fact for novice users. The main control panel shows the source code of the program, highlighting the statement that is currently being animated. On the stage is the animation of a table being sorted with the bubblesort algorithm. Above the table is an animation of a comparison between two values of the table: "YES!" signifies that the comparison returns true.

2.4 Jeliot 2000

The user interface of Jeliot I proved to be difficult for novices. Therefore, a new version of Jeliot was designed and developed by Pekka Uronen during a visit to the Weizmann Institute of Science under the supervision of Mordechai Ben-Ari. Jeliot 2000 [3] was specifically designed to support teaching of novice learners.

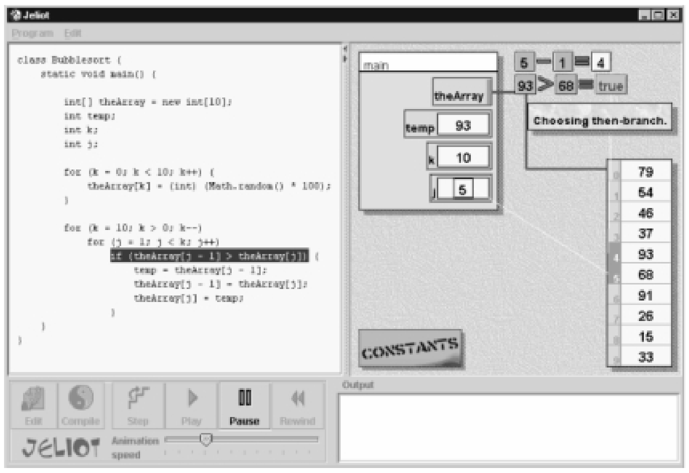

```
class Bubblesort {
    static void main() {

        int[] theArray = new int[10];
        int temp;
        int k;
        int j;

        for (k = 0; k < 10; k++) {
            theArray[k] = (int) (Math.random() * 100);
        }

        for (k = 10; k > 0; k--)
            for (j = 1; j < k; j++)
                if (theArray[j - 1] > theArray[j]) {
                    temp = theArray[j - 1];
                    theArray[j - 1] = theArray[j];
                    theArray[j] = temp;
                }
    }
}
```

Fig. 2. A screenshot of Jeliot 2000

Two principles guided the design of Jeliot 2000: completeness and continuity. Every feature of the program must be visualized; for example, the use of a constant is animated by having the constant move from an icon for a constant store. Moreover, the animation must make the relations between actions in the program explicit; for example, the animation of the evaluation of an expression includes the animation of the evaluation of subexpressions.

As the intended users have little or no experience working with computers, the user interface of Jeliot 2000 is kept as simple as possible, without the customization abilities of Jeliot I. Jeliot 2000 displays two panels: one for the source code, and another for the animation. The execution of the program is controlled through familiar VCR-like buttons (see Figure 2).

The implementation of Jeliot I is based on self-animating data types, but this makes it difficult to implement *visual relations* during evaluation of expressions, parameter passing and control structures. Jeliot 2000 embeds the animation system within an interpreter for Java source code, giving more power of expression at the cost of a more complicated implementation. It was a challenge to produce a smooth animation, because the visual objects that represent an expression must remain displayed for the user to examine in the context of the source code. Jeliot 2000 is written in Java like Jeliot I, but unlike the Web-based client-server architecture of Jeliot I, it is structured as a single application for simplicity and reliability in a school PC laboratory. The current implementation of Jeliot 2000 is limited in the language constructs that it supports.

Figure 2 shows a screen capture from Jeliot 2000. The left panel shows the program code that is animated. The lower-left corner contains the simple VCR-like control panel. The lower-right corner of the user interface contains a textbox that Jeliot 2000 uses to display the output. The animation is performed on the stage in the right panel. Here the animated algorithm is the same bubblesort algorithm as in Figure 1. The lower left corner of the stage displays a "source" of constants. The rest of the stage displays the animation: on the left, a box representing the main method including the variables that are declared within the method. At the moment, a comparison of two values of the table is being animated. One can see from the picture all subexpressions that are needed to evaluate the expression in the program code. In addition, an explanation of the evaluation is displayed.

2.5 Comparison and Discussion

Eliot and Jeliot I were aimed at teaching algorithms and data structures. They are more useful when the student already knows the elements of programming. Constructing of an animation is semi-automatic. Jeliot 2000 was made for novices to illustrate how a Java program works. Animation is fully automatic, and the user is not able to customize the animation. Table 1 lists the main features of the systems.

Comparing the animations of Jeliot I and Jeliot 2000 is difficult because in Jeliot I one can have several adjustable views, while Jeliot 2000 has only one fixed view. Another difference between the systems is in the level of explanation. Jeliot I does not present as many explanatory features as Jeliot 2000; these were added in the development of Jeliot 2000 as essential for novices.

Table 1. Characteristics of the Jeliot family.

	Eliot	*Jeliot I*	*Jeliot 2000*
Language	C	Java	Java
Animated objects selectable	+	+	−
Visual attributes adjustable	+	+	−
Animated data types			
Number	+	+	+
Boolean	−	+	−
Character	+	+	−
Array	+	+	+
Queue	−	+	−
Stack	−	+	−
Tree	+	−	−
Active code line highlighted	−	+	+
Number of stages	many	many	1

Automation is one of the key features of Jeliot. Marc Brown [6] has discussed the problems of automatic animation. He argues that in general there is no one-to-one correspondence between the statements of a program and the images of the animation. An ideal animation according to him also shows synthetic metastructures of the algorithm. Of course, a fully automatic animation system cannot produce any synthetic metastructures. But in some cases they can be achieved by customizing a view, which is possible in Eliot and Jeliot I. And it is always possible to use an automatic system in an incremental way by programming synthetic metastructures as additional data structures of the algorithm and letting the animation system visualize them.

On the level of abstraction at which the programs are executed in Jeliot, informative displays are easy to construct, though the length of the program code and number of the data structures can be a problem. In Eliot and Jeliot I, one can add new stages to accommodate all the data structures that are to be animated. During the development of Jeliot I, this feature was used to debug the system, proving that even a large amount of code can be accommodated. Moreover, one can add data types of one's own inside Jeliot I and so animate even complex data structures. In Jeliot 2000, problems may arise if there are too many data structures to be simultaneously animated because Jeliot 2000 has only one stage. However, Jeliot 2000 was designed for novice users, so this limitation is not important.

3 Empirical Evaluation

There is no question that visualizations and animation of algorithms and programs is appealing, and can increase the motivation of students studying computer science. Intuitively, it would seem that they would significantly improve learning of computer science concepts; unfortunately, empirical studies do not unequivocally support this claim. From its inception, the Jeliot family has been subjected to extensive empirical evaluation; the results clearly show when program visualization can help and when not. In this section, we discuss some theoretical background, briefly describe empirical work by John Stasko, and then present details of the empirical evaluation of Eliot, Jeliot I, and Jeliot 2000.

3.1 When Does Visualization Help?

Petre and Green [15,16] examined the use of visual programming by novices and experts. They concluded that the main advantage of graphics is the information contained in *secondary notation*, which is the informal part of the graphics: placement, color, indentation, and so on. Experts use this information efficiently to understand a graphics display; even if two experts use different secondary notation, they are able to easily decipher each others conventions and to recognize them as the work of experts. Novices ignore or misinterpret secondary notation, so their use of graphics is highly inefficient. Petre and Green conclude that: (a) the notational needs of experts and novices are different, and (b) novices must be explicitly taught to *read* graphics.

In a broader context, Mayer [17] performed a sequence of experiments on multimedia learning. He found that visualizations must be accompanied by simultaneous textual or verbal explanations to be effective. Multimedia guides students' attention and helps them create connections between text and concepts.

These results directly influenced the development of Jeliot 2000, by pointing out the need to a different tool for novices, and the need to include explanatory text with the animation of control structures.

3.2 Stasko's Work

Stasko, Badre and Lewis [18] used algorithm animation to teach a complicated algorithm to graduate students in computer science, but the results were disappointing: the group that used animation did not perform better than the control group. They conjecture that the students had not used animation before and found it difficult to map the graphics elements of the animation to the algorithm. In another experiment, Byrne, Carambone, and Stasko [19] showed that students in the animation groups got better grades on challenging questions for simple algorithms, but on difficult algorithms the differences were not significant. Kehoe, Stasko, and Taylor [20] found that algorithm animation is more effective in open homework sessions than in closed examinations. As one would expect from Mayer's work, they found that animation is not useful in isolation: students need human explanations to accompany the animations.

3.3 Evaluating Eliot and Jeliot I

An empirical evaluation of Eliot was carried out in a course on data structures [21,22], using questionnaires, video tapes, learning diaries and interviews. The studies showed that using Eliot improved the motivation and activation level of the participating students, and that students produced higher quality code and documentation. Jeliot I has also been used for cross-cultural co-operation in teaching programming [23].

Matti Lattu [24] carried out an empirical evaluation of Jeliot I primarily on two groups of high-school students. (A group of university students was also studied, but they did not use Jeliot I in depth.) Semi-structured interviews and observations of the lectures were used. Here is a summary of the results:

– Both students and teachers tend to use continuous execution, rather than step-by-step mode. Some educators might find this result to be counter-intuitive, because step-by-step execution is more interactive and constructivist [25] than continuous execution.
– Teachers frequently used visualization during lectures *before* presenting the program source.
– Jeliot I assisted in concept-forming, especially at the novice level.
– The user interface was too complex for novices. Confirming Petre's claims, the novices had difficulty interpreting the visualization, and the grain of animation was too coarse.

In subsequent research [26], Jeliot I was evaluated for use as a demonstration aid when teaching introductory programming and Java. Several classes were observed during the year and the field notes analyzed. The observation also captured the use of traditional demonstration aids: a blackboard and an overhead projector.

Their first conclusion is that ease and flexibility of use are of paramount importance. Developers of visualization software must ensure that the software is easy to use and reliable; otherwise, low-tech materials will be preferred. Of more interest is the observation that all aspects of a program must be visualized: data, control flow, program code and objects. Jeliot I is primarily a tool for visualizing data, while Jeliot 2000 significantly improved the visualization of control flow. Perhaps the next step is to include visualization of program code and objects. The BlueJ system [27] is an excellent example of this type of visualization tool.

3.4 Evaluating Jeliot 2000

Jeliot 2000 was evaluated by Ronit Ben-Bassat Levy in an experiment [3] that is as close to a controlled experiment as one could hope for: two classes, one using Jeliot 2000 and one as a control group. The experiment was carried out on tenth-grade high school students studying an introductory course on algorithms and programming, and the results were evaluated both quantitatively and qualitatively. The classes were composed randomly, but unfortunately, the control group was better, which made interpretation of the quantitative results somewhat difficult. The experiment was run for a full year, so that the students could overcome the difficulties inherent in using a new system.

The experiment was carried out by testing learning of each new concept as it was studied during the year. In addition, an assignment at the end of the year and a follow-up assignment during the next school year were used to investigate long-term effects. The quantitative test results were supplemented with individual problem-solving sessions which were taped and analyzed. For details of the experimental setup and results, see [3]. We can summarize the results and conclusions as follows:

– The scores of the animation groups showed a proportionally greater improvement, and their average matriculation exam score was the same as that of the control group, even though the latter contained stronger students.
– Mediocre students profit more from animation than either strong or weak students, though the grades of the latter two groups do not suffer.
– Students benefit most if the animation session includes individual instruction.
– The animation group used a different and better *vocabulary of terms* than did the non-animation group. Verbalization is an important step to understanding a concept, so for this reason alone, the use of animation can be justified.
– There was significant improvement in the animation group only after several assignments; one can conclude that it takes time to learn to use an animation tool and to benefit from its use.
– Students from the animation group used a *step-by-step method of explanation*, and some even used *symbols from Jeliot 2000* in order to show the flow of values. Students from the control group expressed themselves in a generalized and verbose manner. This difference in style continued into the next year.

Here is a summary of a problem-solving session on nested if-statements that demonstrates how the above conclusions were arrived at:

– In the control group, only the stronger students could answer the questions, and only after many attempts. They were not sure of the correctness of their answers and had difficulty explaining them.
– The stronger students of the animation group also had difficulties answering this question! They did not use Jeliot 2000 because they believed that they could understand the material without it.
– The weaker students of the animation group refused to work on the problem, claiming that nested if-statements are not legal, or that they did not understand the question.
– The mediocre students of the animation group gave correct answers! They drew a Jeliot 2000 display and used it to hand simulate the execution of the program.

4 Future Plans

The knowledge that has been collected through empirical evaluation of Jeliot has already changed the development of the Jeliot family. Here we present suggestions for the further development of Jeliot.

4.1 Visualization Techniques and User Interface Issues

In automatic program visualization, the animation is performed at constant speed, even though some parts of the program are more difficult to understand than others. The ability to specify varying animation speeds for different parts of the program would make it easier to concentrate on difficult parts of the program. For example, initialization could be run at a higher speed than the statements in the inner loops of a sorting algorithm. The question arises: How does the user specify such difficult parts? The user would have to specify such parts through special comments in the program or using the user interface. The next paragraph suggests that semi-automated visualization could help with this specification.

For a novice user who has never programmed, automatic animation is essential. However, as the user becomes more experienced, he or she will want to control the configuration of the animations, for example, to select the speed of animation of different parts of the program, or even to skip the animation of parts like initialization. The user will also want to configure the visual elements for color, form and placement as was done in Jeliot I. Jeliot I also showed that storing configurations is important, because it fosters reuse animations, making them easier to share between a teacher and a student, or among the students themselves.

While forcing users to shift their gaze from one point to another on the screen is not recommended [28], it is important to guide the user in focusing on significant elements of the animation. One possibility would be to use sound [17]: the user would come to recognize specific sounds as guiding focus to specific locations.

4.2 Visualization with Jeliot

Structures of the programming language. The animation of method calls and array access is not entirely transparent in any of the systems of the Jeliot family, even though precisely these elements can be difficult for novices. It is important to find better ways to illustrate how a method gets its parameters and how multi-dimensional arrays are accessed. New self-animating data types such as lists and graphs would extend the applicability of Jeliot. Furthermore, Jeliot should make it easy for the user to create a new self-animating data type.

Object-oriented programming. Jeliot uses Java, a popular object-oriented language, but it can not handle objects or user defined classes. The next version of the Jeliot should provide better support for animating aspects of objects, such as object creation and method calling. Jeliot is quite good at animating the dynamic aspects of program execution, but to support object-oriented programming, it should also visualize the class structure in order to show the *uses* and *inherits from* relationships among the classes (cf. [27]).

Other programming languages. Currently Jeliot supports only programs written in the Java language. A visual debugger for Scheme [29] was implemented by slightly modifying Eliot, showing that the Jeliot could be modified to support other programming languages.

Visualizations of other subjects. Many dynamic phenomena can be described as algorithms, and therefore visualized by Jeliot [30]. For example, it would be possible to visualize the Mendelian rules of inheritance for a biology class.

4.3 Integration with Other Environments

We would like to integrate Jeliot into the learning environment so that metadata [31] could be collected about the students. For example, if Jeliot could collect metadata about the difficulties that an individual student has, this could be used both by the teacher and by Jeliot itself to adapt the animation speed as described above.

Jeliot could be integrated with other program visualization tools such as BlueJ [27] to provide a richer variety of views of the program and its execution.

A natural application of automatic animation is debugging [1]. The debugging abilities of Jeliot could be improved by implementing all of the Java language, and also by more efficient highlighting of the code. Errors found during compilation should be highlighted and partial animation performed if possible. Perhaps even common syntax errors could be animated. Thus, Jeliot could form part of a semi-automated assessment system for programming exercises [32].

4.4 Jeliot in a Learning and Development Community

Users should be able to interact with each other. Jeliot could be integrated with Internet communication tools to facilitate students working together on the same project, by enabling all participants to view the same animation.

Many of the proposed extensions to Jeliot could be implemented independently. The Jeliot source code could be licensed as free software, perhaps under the GNU general public license (GPL), with coordination coming from the Department of Computer Science at the University of Joensuu.

5 Conclusion

The phases of the history of Jeliot reflect different trends or approaches in computer science education, especially in teaching how to program. The predecessor Salsa was a package of ready-made animations for teaching string algorithms: it emphasized the instructive perspective. The Eliot, Jeliot I, and Jeliot 2000 systems, with their fully or semi-automatic animation tools, are examples of constructive learning environments. The future platforms will be worked out by extended and networked teams: they represent the idea of a learning community.

In these communities, one can no more make a distinction between a teacher, a learner, and a designer.

One of the main lessons learned during the development and evaluation cycle of Jeliot is that of different learners and learner groups. An animation system should always offer a solution to a certain learner group's needs. Therefore, an evaluation is not just another stage in the design and implementation of an environment, but should be carried out simultaneously during the whole process. Moreover, there is seldom one single best application for all animation or program comprehension needs, but rather a bunch of components of which a learner can pick up the ones she needs.

To sum up, what we have learned during the close to ten years of working with program animation, is that animation as well as apparently other learning tools should help a learner at his individual learning difficulties adaptively, distance independently, and taking into account diverse learning and cognitive styles.

References

1. Lahtinen, S., Sutinen, E., Tarhio, J.: Automated animation of algorithms with Eliot. J. Visual Languages and Computing **9** (1998) 337–349.
2. Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., Vanninen, P.: Animation of user algorithms on the Web. In: Proceedings of VL' 97 IEEE Symposium on Visual Languages. (1997) 360–367.
3. Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P.A.: The Jeliot 2000 program animation system. Journal of Visual Languages and Computing (2001 (submitted)) Preliminary version in E. Sutinen (ed.) *First Program Visualization Workshop*, pages 131–140, University of Joensuu, 2001.
4. Sutinen, E., Tarhio, J., Teräsvirta, T.: Easy algorithm animation on the Web. Multimedia Tools and Applications (2001) (to appear).
5. Sutinen, E., Tarhio, J.: String matching animator Salsa. In Tombak, M., ed.: Proceedings of Third Symposium on Programming Languages and Software, University of Tartu (1993) 120–129.
6. Brown, M.: Perspectives on algorithm animation. In: Proceedings of CHI '88. (1988) 33–38.
7. Myers, B.: Incense: a system for displaying data structures. ACM Computer Graphics **17** (1983) 115–125.
8. Moher, T.: Provide: a process visualization and debugging environment. IEEE Transactions on Software Engineering **14** (1988) 849–857.
9. Müller, H., Winckler, J., Grzybek, S., Otte, M., Stoll, B., Equoy, F., Higelin, N.: The program animation system pastis. Journal of Visualization and Computer Animation **2** (1991) 26–33.
10. Henry, R., Whaley, K., Forstall, B.: The University of Washington illustrating compiler. In: Proceedings of ACM SIGPLAN '90 Symposium on Compiler Construction. Volume 25(6) of SIGPLAN Notices. (1990) 223–233.
11. Mukherjea, S., Stasko, J.: Toward visual debugging: integrating algorithm animation capabilities within a source level debugger. ACM Transactions on Computer-Human Interaction **1** (1994) 215–244.
12. Baeza-Yates, R., Fuentes, L.: A framework to animate string algorithms. Information Processing Letters **59** (1996) 241–244.

13. Sanders, I., Harshila, G.: AAPT: Algorithm animator and programming toolbox. SIGCSE Bulletin **23** (1991) 41–47.
14. Stasko, J.: Polka Animation Designer's Package. (1994) Animator's Manual, included in Polka software documentation.
15. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM **38** (1995) 33–44.
16. Petre, M., Green, T.R.: Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. Journal of Visual Languages and Computing **4** (1993) 55–70.
17. Mayer, R.E.: Multimedia learning: Are we asking the right questions? Educational Psychologist **32** (1997) 1–19.
18. Stasko, J., Badre, A., Lewis, C.: Do algorithm animations assist learning: An empirical study and analysis. In: Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, Amsterdam, The Netherlands (1993) 61–66.
19. Byrne, M., Catrambone, R., Stasko, J.: Do algorithm animations aid learning? Technical Report GIT-GVU-96-19, Georgia Institute of Technology (1996).
20. Kehoe, C., Stasko, J., Taylor, A.: Rethinking the evaluation of algorithm animations as learning aids: An observational study. Technical Report GIT-GVU-99-10, Georgia Institute of Technology (1999).
21. Sutinen, E., Tarhio, J., Lahtinen, S.P., Tuovinen, A.P., Rautama, E., Meisalo, V.: Eliot—an algorithm animation environment. Technical Report A-1997-4, University of Helsinki (1997). http://www.cs.helsinki.fi/tr/a-1997/4/a-1997-4.ps.gz.
22. Meisalo, V., Sutinen, E., Tarhio, J.: CLAP: teaching data structures in a creative way. In: Proceedings Integrating Technology into Computer Science Education (ITiCSE 97), Uppsala (1997) 117–119.
23. Järvinen, K., Pienimäki, T., Kyaruzi, J., Sutinen, E., Teräsvirta, T.: Between Tanzania and Finland: Learning Java over the Web. In: Proceedings Special Interest Group in Computer Science Education (SIGCSE 99), New Orleans, LA (1999) 217–221.
24. Lattu, M., Meisalo, V., Tarhio, J.: How a visualization tool can be used: Evaluating a tool in a research and development project. In: 12th Workshop of the Psychology of Programming Interest Group, Corenza, Italy (2000) 19–32. http://www.ppig.org/papers/12th-lattu.pdf.
25. Ben-Ari, M.: Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching **20** (2001) 45–73.
26. Lattu, M., Meisalo, V., Tarhio, J.: On using a visualization tool as a demonstration aid. In Sutinen, E., ed.: First Program Visualization Workshop, University of Joensuu (2001) 141–162.
27. Kölling, M., Rosenberg, J.: Guidelines for teaching object orientation with Java. In: Proceedings Integrating Technology into Computer Science Education (ITiCSE 01), Canterbury, UK (2001) 33–36. www.bluej.org.
28. Saariluoma, P.: Psychological problems in program visualization. In Sutinen, E., ed.: Proceedings of the First Program Visualization Workshop. Volume 1 of International Proceedings Series., Department of Computer Science, University of Joensuu (2001) 13–27.
29. Lahtinen, S.P.: Visual debugger for Scheme. Master's thesis, Deparment of Computer Science, University of Helsinki (1996) (in Finnish).

30. Meisalo, V., Sutinen, E., Tarhio, J., Teräsvirta, T.: Combining algorithmic and creative problem solving on the web. In Davies, G., ed.: Proceedings of Teleteaching '98/IFIP World Computer Congress 1998, Austrian Computer Society (1998) 715–724.
31. Markus, B.: Educational metadata. In: Proceedings of Qua Vadis-International. FIG Working Week, Prague (2000).
32. Higgins, C., Suhonen, J., Sutinen, E.: Model for a semi-automatic assessment tool in a web-based learning environment. In Lee, C.H., ed.: Proceedings of ICCE/SchoolNet 2001 Conference, Seoul, Korea (2001) 1213–1220.

Animating Algorithms Live and Post Mortem

Stephan Diehl, Carsten Görg, and Andreas Kerren

University of Saarland,
FR 6.2 Informatik,
PO Box 15 11 50,
D-66041 Saarbrücken, Germany
{diehl,goerg,kerren}@cs.uni-sb.de

Abstract.

We first give an overview of the features of the GANIMAL Framework introducing several new concepts not present in any previous algorithm animation system. Then we focus on its mechanisms for mixing live and post mortem visualization which are in particular very useful for algorithms which restructure graphs.

1 Introduction

In recent years we have developed several educational software systems for topics in compiler design and theoretical computer science [1,13]. These systems have in common that they teach computational models by animating computations of instances of these models with example inputs.

In the project GANIMAL we develop generators, which produce interactive visualizations and animations of different compiler phases. The generators form the basis of new kinds of exercises as part of educational software [9,8]. The learner can focus on certain aspects in the generated, interactive animation and see what effects small modifications in the specification have. With the help of such observations he formulates hypotheses and checks these empirically. The learning software does not act as an anonymous, all-knowing authority which shows his errors. Instead, our approach offers a way for explorative, self-controlled learning. Such a visual experimental approach is not meant to replace, but to enhance classical teaching of theoretical contents.

To ease the creation of interactive animations we developed the GANIMAL Framework. The GANIMAL Framework and in particular the language GANILA provide a powerful set of features. It integrates concepts of different classical systems: Interesting events and views (BALSA [3]), step-by-step execution and breakpoints (BALSA-II [2]), and parallel execution (TANGO [15]). In addition it offers new features like alternative interesting events and alternative code blocks, visualization of invariants for program points and blocks, foresighted graphlayout, and mixing of post mortem and live/online algorithm animation which is a prerequisite for visualization control of loops and recursion, i.e. the ability to visualize only the execution of certain program points, e.g. the last five executions of a loop or every second invocation of a recursive method.

This paper is organized as follows. Section 2 introduces the GANIMAL framework, i.e. the software architecture and the basic workflow for creating animations. Section 3 describes the annotations provided by the GANILA language for live algorithm animation and Section 4 describes those for inserting post mortem visualizations into live animations and based on this the visualization control for loops and recursion. As an example we compare in Section 5 live and mixed mode animations of an algorithm which computes least upper bounds. Section 6 concludes.

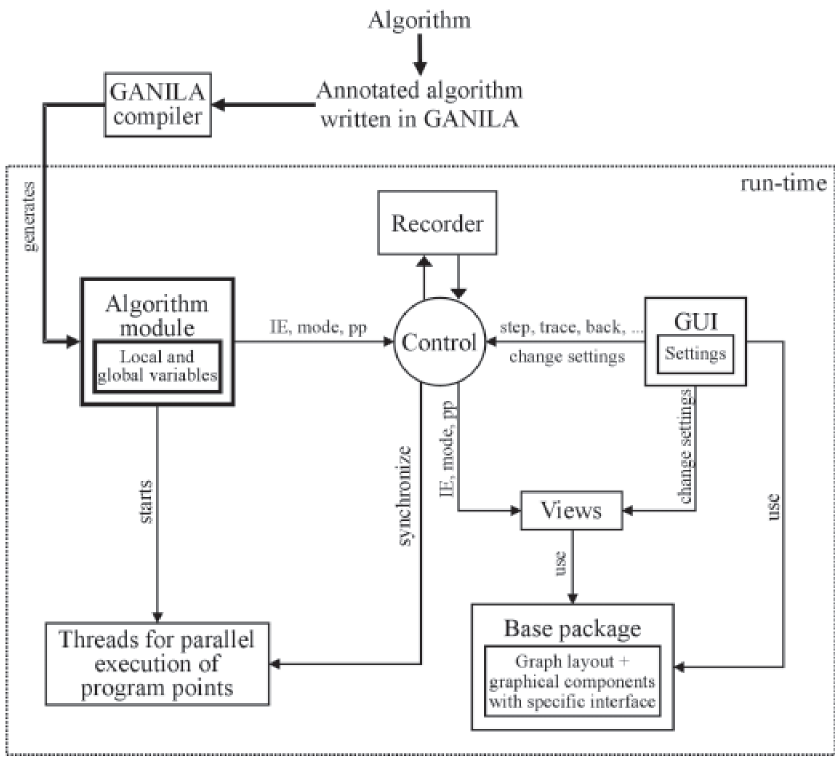
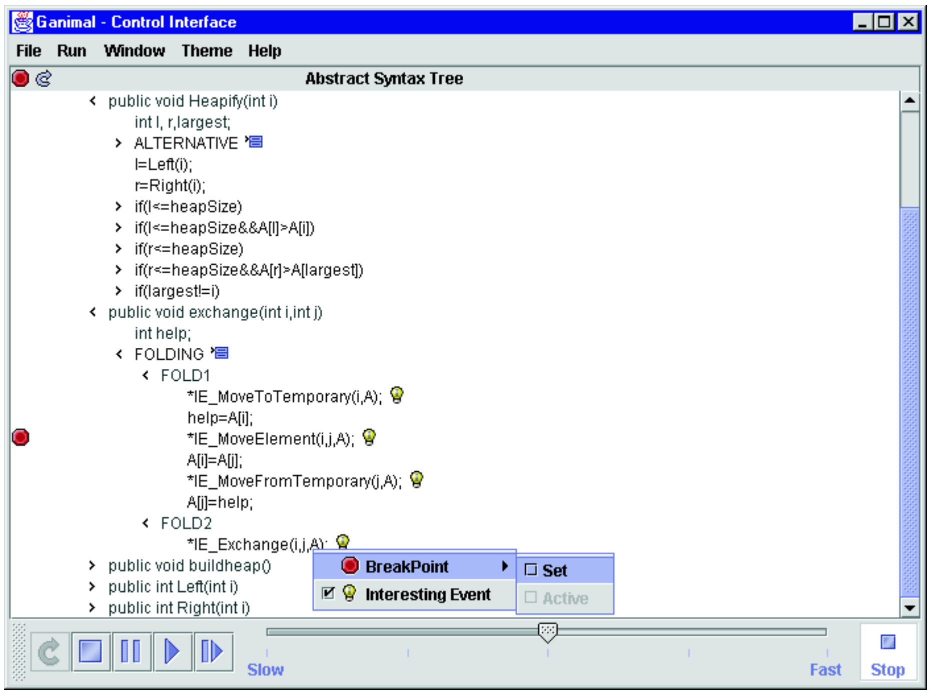


Fig. 1. The GANIMAL Framework

2 The GANIMAL Framework

Based on GANIMAM [10] and the experiences found in related work [6,16] we designed the GANIMAL Framework, see Figure 1. It consists of the GANILA Compiler and a runtime system. The compiler generates code which in combination with the runtime system produces the interactive animations. More precisely, given a specification written in the language GANILA, the generator produces an algorithm module and the initial settings, i.e. meta-information associated with each program point of the algorithm. During the execution of the algorithm this module sends interesting events (`IE`) containing the current program point `pp` and the current animation mode (RECORD,PLAY) to a control object. The control object checks the settings for this program point. If the interesting event has not been deactivated at this program point, it is send to all views. Each view can have its own settings and decide whether it will invoke its event handler for this interesting event. Based on the animation mode the event handling routines produce graphical output or simply change some internal state and defer the graphical output until the mode is set to PLAY. At runtime the graphical user interface (Figure 2) can be used to change the settings of each program point.

All views should use the base package which consists of a set of Java classes providing primitive methods for communication, graphical output, and animation. The use of the base package fosters a consistent look-and-feel of different views.



```
< public void Heapify(int i)
    int l, r,largest;
  > ALTERNATIVE
    l=Left(i);
    r=Right(i);
  > if(l<=heapSize)
  > if(l<=heapSize&&A[l]>A[i])
  > if(r<=heapSize)
  > if(r<=heapSize&&A[r]>A[largest])
  > if(largest!=i)
< public void exchange(int i,int j)
    int help;
  < FOLDING
    < FOLD1
        *IE_MoveToTemporary(i,A);
        help=A[i];
        *IE_MoveElement(i,j,A);
        A[i]=A[j];
        *IE_MoveFromTemporary(j,A);
        A[j]=help;
    < FOLD2
        *IE_Exchange(i,j,A);
> public void buildheap()
> public int Left(int i)
> public int Right(int i)
```

At run-time the user can set break points, select alternative events or alternative code blocks, activate or deactivate interesting events, and select parallel or sequential execution of certain blocks. Furthermore he can control the animation using a VCR like control to start, pause, or step through the animation.

Fig. 2. Graphical User Interface

3 Annotations for Live Animation

The language GANILA extends Java by interesting events, parallel execution of program points, recording and replaying mechanisms (e.g. by foresighted graph layout), break- and backtrack points, and declarations to import views. The compiler called GAJA translates GANILA into Java. For every annotated program point its annotation can be activated and deactivated at run time using a graphical user interface. The resulting settings can be defined for the whole animation, as well as individually for each view.

3.1 Predefined Views

Our system provides a set of predefined views which can be imported into a GANILA program using `view <Name>(Parameter)`.

```
// A view without parameters
view CodeView();
// A view with parameters
view SoundView("http://www.cs.uni-sb.de/sounds/");
// An algorithm-specific view
view HeapsortView();
```

The developer of the animation can simply use these views or extend their functionality using inheritance. The methods of the views are event handling routines. In the example above `HeapsortView` is such a newly created view.

HTMLView: In GANILA it is possible to associate program points with web pages. A third party component integrated into the system (IceBrowser-Java-Bean) allows to show HTML content in a view. Moreover it is possible to transfer runtime data, which is accessible at the program point, to a CGI-Script on a server. The server can thus produce context-sensitive HTML pages. In "Literate Programming" [14] a static document is produced from the documentations at different program points in the source code. In contrast, in GANILA documentation can be shown whenever the program point is reached during execution.

GraphView: The GraphView provides several algorithms to layout a graph, see Section 4.2 for more details. Nodes and edges can be added or removed. Here almost all kinds of Java SWING components can be used as nodes.

CodeView: The CodeView shows a textual representation of the program executed and highlights the current program point.

SoundView (Aura): Analogous to the HTMLView, this view associates program points with sound files, e.g. containing spoken explanations. As part of an interesting event it receives the URL of a sound file. Starting, stopping, and repeating the play-back, as well as its volume can be controlled by interesting events.

3.2 Interesting Events

The following excerpts show the GANILA specification of a simple operation, which is used as an example by various algorithm animation systems: the swapping of the content of two elements of an array, here `a[i]` and `a[j]`.

```
help = a[i]; *IE_MoveToTemporary(i);
a[i] = a[j]; *IE_MoveElement(i,j);
a[j] = help; *IE_MoveFromTemporary(j);
```

Interesting events have the prefix `*IE_` and transfer local information in their arguments to the different views. Obviously in such a view the value of `a[i]` could be moved to a representation of the auxiliary variable. Then the value of `a[j]` would be moved to `a[i]` and finally the value of the auxiliary variable would be moved to `a[j]`. So far, the GANILA events (GEvents) work very much like those in other multi-view event-based algorithm animation systems like ZEUS [4]. One important difference is that such a GEvent is first send to the control of the framework and is subject to the settings like every program point. As a consequence the user can activate or deactivate the effect of an interesting event at run time using the GUI, see Figure 2. The event handlers of each view must be programmed such that they actually receive a deactivated event and they might even change the internal state of the view to prevent inconsistencies, but it should not produce any visual output. Every view registers with the control object which in turn forwards each event to all registered views. An event handler can even create new views which it can register with the control object. The algorithm object, the control object, and the views are implementing the MVC design pattern (model, view, control) which is a combination of the Observer, Composite, and Strategy patterns [11].

3.3 Alternative Interesting Events and Alternative Blocks

GANILA also supports the grouping of program points by enclosing them in `*{` and `*}` to from a block. The statement `*FOLD *{ <Eventlist> *}` triggers one or more alternative GEvents for a program point or block. The following example also shown in the GUI in Figure 2 illustrates the use of this statement:

```
public void exchange(int i, int j) {
    int help;
     *{  help = A[i]; *IE_MoveToTemporary(i,A);
         A[i] = A[j]; *IE_MoveElement(i,j,A);
         A[j] = help; *IE_MoveFromTemporary(j,A);
      *}
     *FOLD *{ *IE_Exchange(i,j,A); *}
}
```

Using the GUI the user can decide at run time whether the events in the block or the alternative event is triggered. In both cases the program code in the block is executed. By selecting the alternative event the views could move the two values of the field in parallel to their new positions. Note that in this solution the event handler could use concurrency internally. This is completely different from using the parallel operator as discussed in the next section.

The `*FOLD` construct is meant to support semantical zooming, i.e. in many cases the events in the block, in particular if other methods are invoked, will produce more fine grained animations than the alternative events.

In contrast to `*FOLD` the GANILA construct `*ALT` allows the programmer to provide two different program blocks which should produce the same results. The user can then decide in the GUI which of these program blocks should be actually executed.

```
int min;
*{ min=a[0];
   for(int i=1;i<a.length;i++)
    { *IE_Compare(a,i,min);
      if (a[i]<min) min=a[i];
    }
*}
*ALT
*{ min=a[a.length];
   for(int i=a.length;i>=0;i--)
    { *IE_Compare(a,i,min);
      if (a[i]<min) min=a[i];
    }
*}
```

3.4 Parallel Execution

Using the operator `*||` two program points or blocks can be executed in parallel.

```
    *{ *IE_AssignTemporary(1,i); help1 = a[i]; *}
*|| *{ *IE_AssignTemporary(2,j); help2 = a[j]; *}

    *{ *IE_MoveTemporary(j,1); a[j] = help1; *}
*|| *{ *IE_MoveTemporary(i,2); a[i] = help2; *}
```

In the above program first the two assignments to `help1` and `help2`, as well as the respective events are executed in parallel, then the two assignments to `a[i]` and `a[j]` and the respective events are executed in parallel. As a result the corresponding animations run in parallel. Note that if we would use a single auxiliary variable, data dependencies make parallel execution impossible. In other words, the algorithm had to be slightly changed to enable the parallel animations. The parallel operator automatically creates, starts and synchronizes Java threads for each of the two blocks.

3.5 Test of Invariants

To understand an algorithm it is often necessary to look at properties, which are true for all program states at certain program points. In our framework the developer of an animation can provide a hypothesis and have it checked at certain program points. In the following example the so-called heap property is checked for a part of the heap sort algorithm:

```
*IV(a[i]>=a[2*i+1] && a[i]>=a[2*i+2])
*{
   // part of the heap sort algorithm
*}
```

If the expression is an invariant of a program point, then it should never yield `false` when this program point is executed. If a block is annotated with such an expression, the user will see which program points change the program state such that the invariant is violated, and which program points reestablish the invariant. In addition the user can formulate hypotheses at run time and have them tested by the system. In doing so it is sometimes necessary to invoke complex functions, which have been programmed by the developer of the animation. As it does not make sense to enable the user to invoke every function of the program, the developer can annotate those functions with `interactive` which should be accessible through the GUI at run time.

```
interactive boolean heapProperty(a,i) {
   // checks the heap property
   return a[i]>=a[2*i+1] && a[i]>=a[2*i+2];
}
```

Invariant visualization in GANILA is an example of state mapping [5], i.e. the visualization is not triggered at certain program points through events, but the view has direct access to the program state and automatically adapts its visualization whenever the state changes.

3.6 Break and Backtrack Points

Program points can be marked with `*BREAK` in the GANILA code or through the GUI at run-time as break points. When the execution of the algorithm reaches this program point, the execution is paused and the user can investigate the current state, continue with the animation, or trace it step-by-step.

Backtrack points are marked with `*SAVE`. When the execution of the algorithm reaches such a program point, the current state is copied to the history. Backtrack points are a means to implement reverse execution of the algorithm or repeated execution from a certain point with changed settings. Another way to repeat the execution is to replay all interesting events. This is a more time-consuming, but less memory-consuming alternative provided by the system.

4 Mixing Live and Post Mortem Visualization

In addition to sending events to all registered views the control object can record all events and resend them later. In this case the event handling routines of each view produce no graphical output, but can change some internal state and defer the graphical output until the event is resend. In this section we look at those constructs of GANILA which enable mixing of live and post mortem visualization.

4.1 RECORD/REPLAY

The GANILA code below shows how to annotate the algorithm to enable post mortem visualization. In

```
*RECORD;
  // annotated program code, e.g.
  // for the generation of an NFA from a regular expression
*REPLAY;
```

By default algorithms are executed in PLAY mode. In this mode all interesting events are immediately executed. The instruction `*RECORD` selects the RECORD mode. In this mode all interesting events are not executed, but stored by the control object in their dynamic order. The instruction `*REPLAY` first executes all recorded events. Then it switches into PLAY mode.

Many naive post-mortem visualization systems work like this. They just replay recorded events. Although they actually know the whole story before they even draw the first line, they do not exploit this fact to improve the visual output.

To enable views to interpret interesting events being fully aware of what events will occur next, the control also forwards events in RECORD mode to all views, but the views are only allowed to modify their internal state, but no graphical output must be produced. This must be deferred until the recorded events are resend.

The `*RECORD/*REPLAY` mechanism allows to mix post mortem and life/online algorithm animation. This is a feature not present in any of the algorithm animation systems we are aware of.

4.2 Foresighted Graphlayout

Often animations of algorithms which change graphs are confusing because they add or remove nodes and edges, and as a consequence the layout of the whole graph is recomputed. In the new layout nodes are drawn at new positions, and a smooth animation called morphing moves nodes from their old to their new positions. Such animations are often nice to look at, but for the user it is not apparent which modifications are due to the animated algorithm and which are due to the drawing algorithm. GANILA supports mechanisms for foresighted

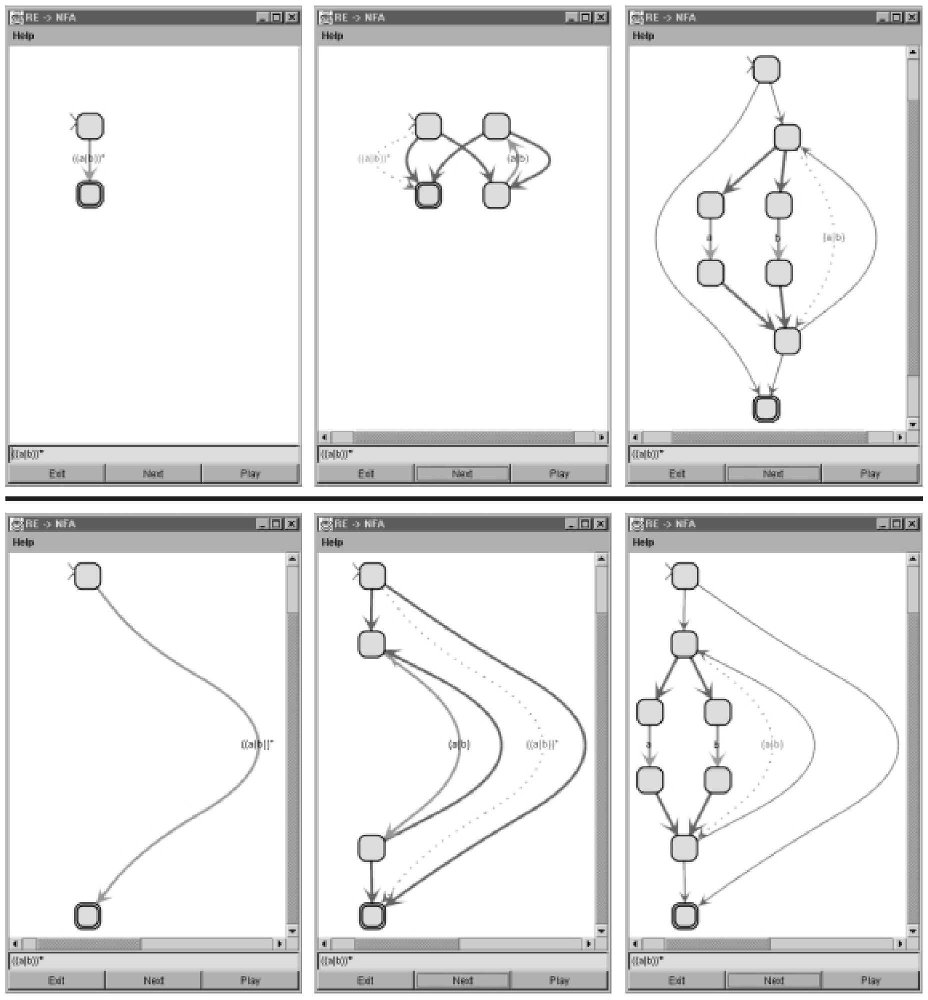


Fig. 3. Ad-Hoc (upper row) and Foresighted Graphlayout (lower row) animating the generation of finite automata

layout, i.e. a graph is drawn exploiting information about subsequent changes of the graph [7].

Figure 3 illustrates how the mechanism can be used to animate the generation of a nondeterministic finite automaton from a regular expression (RE→NFA). The GraphView automatically uses Foresighted Layout when events are recorded and replayed. In the upper row three generation steps are shown using a usual graph drawing algorithm; in the row below Foresighted Layout is used. Without Foresighted Layout it is difficult to see which nodes and edges are added or removed at each step.

4.3 Controlling the Visualization of Loops and Recursion

Often interesting events are placed within loops or recursive method invocations, e.g. when a list is traversed by an iterative sorting algorithm like insertion sort or a recursive sorting algorithm like Quicksort. If the iteration or recursion is part of a larger algorithm, it can be annoying that all iterations or invocations are visualized. For the user it could be very boring to watch 100 iterations and it could be sufficient for understanding the algorithm to see just the last three iterations. Our solution to this problem is based on recording all and replaying only certain events at the end of the loop or recursion. To enable such a selective visualization GANILA allows to annotate Java's loop statements (`do`, `while`, `for`) with visualization conditions. These are written within brackets following the loop condition:

```
for(int j=0;j<100;j++) [$i>=$n-5] { foo(j); }
```

The animation of the execution of the above example program will only visualize the last five invocations of the function `foo()`. Here the variable `$i` denotes the number of the current iteration and the variable `$n` the maximal number of iterations of the respective loop. Note that both values can only be computed at run time.

Analogous to the annotation of loops recursive method invocations can be annotated. Here the variable `$i` represents the current depth and the variable `$n` the maximal depth of the recursion.

Animation control for loops and recursion first records all events until the last iteration or recursion is reached. Then it know the value of `$n` and can resend the relevant events.

5 Example: Animating the Computation of Least Upper Bounds

To illustrate the advantages of mixing live and post mortem visualization we look at an algorithm for computing a complete semi-lattice given a set of pairs of integers. A complete semi-lattice contains for each two pairs (a, b) and (a', b') their least upper bound $(\mathsf{max}(a, a'), \mathsf{max}(b, b'))$. An example animation for the set $\{(2, 1), (3, 1), (1, 4)\}$ is shown in Figure 4. After step 10 the user adds interactively the pair $(1, 2)$ to the initial set of pairs. To produce the animation we record all events before the user interaction. Then we replay these using adhoc (upper row) or Foresighted (lower row) Layout. Now the user sees the actual state (step 10) and can change the state before the animation continues. In this example we actually only need the simplest version of foresighted layout. In the adhoc layout at almost every step nodes and edges change their positions; intermediate morphing animations help the user keep track of the mental map. Using Foresighted Layout this is only the case between step 10 and 11, because we cannot foresee the result of the user interaction. At all other steps no position changes of nodes and edges take place. At step 21 only an edge between the pair

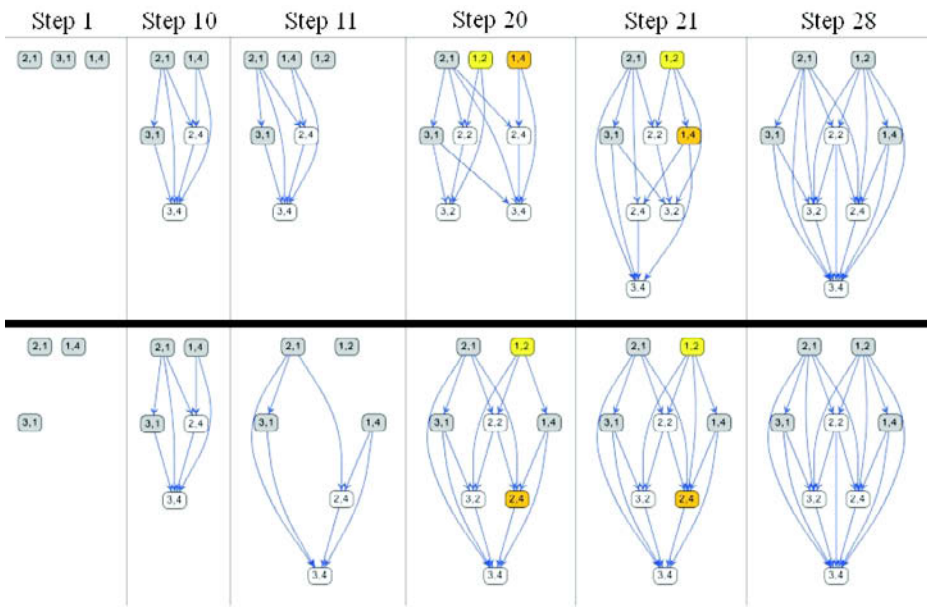


Fig. 4. Adhoc (upper row) and Foresighted Graphlayout (lower row) animating the computation of least upper bounds

(1, 2) and (2, 4) is added. As a consequence adhoc layout changes the position of almost every node, whereas Foresighted Layout just adds this edge.

6 Conclusion

A prototypical implementation of the compiler, as well as interactive animations, which have been produced by the compiler (e.g. heap sort, and the generation and computation of finite automata) are available. More information about the GANIMAL project, as well as more examples can be found online [12].

Acknowledgement. This research has been partially supported by the German Research Council (DFG) under grant WI 576/8-1 and WI 576/8-3.

References

1. B. Braune, S. Diehl, A. Kerren, and R. Wilhelm. Animation of the Generation and Computation of Finite Automata for Learning Software. In *Proceedings of Workshop on Implementing Automata*, volume Springer LNCS 2214, Potsdam, 2001.
2. M. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.

3. M. Brown and R. Sedgewick. A system for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984.
4. M. H. Brown. Zeus: A System for Algorithm Animation and Multiview Editing. In *IEEE Workshop on Visual Languages*, pages 4–9, 1991.
5. Camil Demetrescu, Irene Finocchi, and John Stasko. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
6. Eds.: S. Diehl and A. Kerren. Proceedings of the GI-Workshop "Software Visualization" SV2000. Technical Report A/01/2000, FR 6.2 - Informatik, University of Saarland, May 2000. `http://www.cs.uni-sb.de/tr/FB14`.
7. S. Diehl, C. Görg, and A. Kerren. Preserving the Mental Map using Foresighted Layout. In *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization VisSym'01*, 2001.
8. S. Diehl and A. Kerren. Increasing Explorativity by Generation. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000.* AACE, 2000.
9. S. Diehl and A. Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001.* ACM, 2001.
10. S. Diehl and T. Kunze. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems*, 16(7), 2000.
11. Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.
12. Ganimal. Project homepage. `http://www.cs.uni-sb.de/GANIMAL`, 2000.
13. A. Kerren. Animation of the Semantical Analysis. In *Proceedings of 8. GI-Fachtagung Informatik und Schule INFOS99 (in German)*, Informatik aktuell. Springer, 1999.
14. D. E. Knuth. *Literate Programming.* Center of the Study of Language and Information - Lecture Notes, No. 27. CSLI Publications, Stanford, California, 1992.
15. J. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9), 1990.
16. J. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997.

Visualising Objects: Abstraction, Encapsulation, Aliasing, and Ownership

James Noble

Computer Science
Victoria University of Wellington.
New Zealand
kjx@mcs.vuw.ac.nz

Abstract.

Each object in an object-oriented program can correspond to one abstraction in the program's design. This correspondence makes visualising object-oriented programs easy: simply render each object in terms of its corresponding abstraction. Unfortunately, the endemic aliasing within object-oriented programs undermines this scheme, as an object's state can depend on the transitive state of many other objects, which may be unknown to the visualisation system. By analysing programs to determine the extent of aliasing, we can construct visualisations to display aliasing directly, and can provide support for more abstract visualisations.

1 Abstract Program Visualisation

Consider the program visualisation shown in Fig. 1. This is a very simple visualisation of a collection abstraction, a sequence of some kind, showing the elements in the sequence but no information about they way the collection is implemented. These kinds of views are very simple to construct for programs written in an object-oriented programming language. According to the *Abstraction, Program, Mapping, Visualisation* or APMV model of visualisation:

> The *pictures* we need to draw correspond to the *abstractions* in the *design* which are the *objects* in the *program* [33].

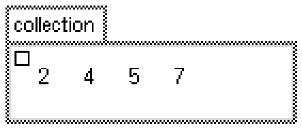


Fig. 1. A Sequence Visualisation

This correspondence between abstractions in the target program's design and objects in the program's implementation provides the main advantage of the APMV model: that the target program's design abstractions can be visualised without the program being modified or annotated, or the implementation details of abstractions being exposed to the visualisation system. An APMV visualisation system produces abstract visualisations *top down* — working from explicit definitions of abstractions in the program rather than their implementations. This is in contrast to most other software visualisation techniques, which work *bottom up*, using programmer supplied annotations, [7,43], procedures [27] or mapping rules [25,11,12] to extract abstractions from their implementations [32].

This technique, as embodied in the Tarraingím (from the Gælic *to draw*) visualisation system [32,33,34,29] can produce both algorithmic visualisations and data structure visualisations. This is because the abstractions making up a program's design, and the objects implementing those abstractions, may be algorithms, data structures, or any combination of the two. Again, this is in contrast to most other program visualisation techniques, which tend to be biased towards producing either data structure or algorithmic visualisations.

For example, Fig. 2 shows how Tarraingím can visualise a simple object (a set implemented using a binary tree, in fact the treeset object from the SELF programming language library [46]) in a number of different ways at different levels of abstraction. Clockwise from the centre, this figure shows an iconic object browser, where each icon represents the contents of one of the set object's variables, bar graph and textual views of the set as a collection, a view of the set as an sequencable collection (the elements are sorted), and a view of the underlying tree implementation.

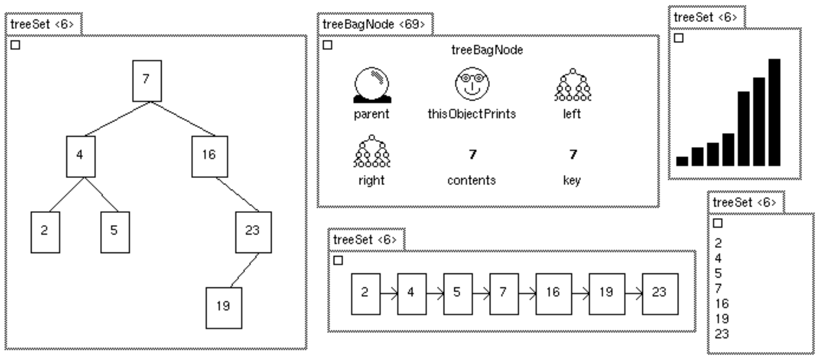
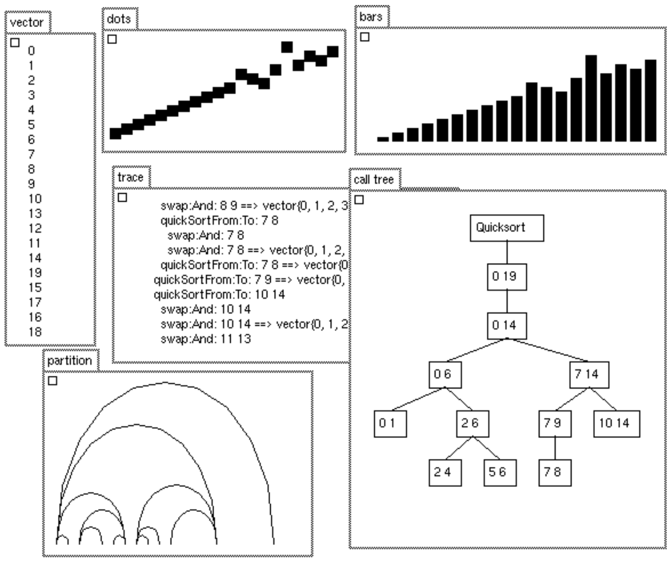


Fig. 2. Tarraingím abstract views

Fig. 3 shows a number of Tarraingím visualisations of the Quicksort algorithm. Clockwise from the centre, this figure shows classic BALSA style dots and bars views of the array being sorted, the recursive call tree of Quicksort invocations, Quicksort's partitions (a curve connects the bounds of each partition with the elements imagined across the bottom of the view), a simple vector view of the elements, and a trace of the algorithm's operations.



```
0
1
2
3
4
5
6
7
8
9
10
13
12
11
14
19
15
17
16
18
```

```
    swap:And: 8 9 ==> vector{0, 1, 2, 3
    quickSortFrom:To: 7 8
      swap:And: 7 8
      swap:And: 7 8 ==> vector{0, 1, 2,
    quickSortFrom:To: 7 8 ==> vector{0
  quickSortFrom:To: 7 9 ==> vector{0,
  quickSortFrom:To: 10 14
    swap:And: 10 14
    swap:And: 10 14 ==> vector{0, 1, 2
    swap:And: 11 13
```

Fig. 3. Tarraingím Quicksort views

Tarraingím is written in the prototype-based object-oriented language Self, and uses the features of that language to access and monitor the objects making up the program [29]. In Tarraingím, abstract data is retrieved from objects top down, by sending accessor messages through objects' interfaces which return that information. Because the data returned is abstract and independent of a particular implementation, the program visualisation system does not have to read the target program's memory directly, and does not have to reinterpret any implementation data structures. Similarly, because objects' implementations are encapsulated behind their interfaces, all operations upon abstractions must be performed through their interfaces. By monitoring the object's public operations, Tarraingím is assured of monitoring all the computation performed in the context of the design abstraction which that object represents.

Fig. 4 illustrates top-down visualisation of a collection that is implemented by a linked list. The view sends messages to the list object to access data, and monitors the messages the list receives from other objects in the program. The view is unaware of the list's implementation, in particular its internal link objects. The list object could be changed (say to a vector that stores elements using an internal array) without affecting the visualisation, providing the list and vector supports the same collection interface.

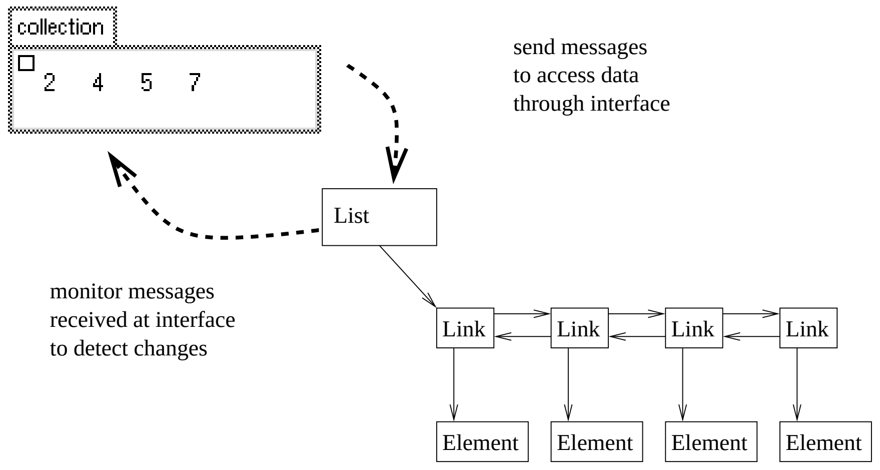


Fig. 4. Top Down Visualisation

Top down visualisation has many advantages. It can produce high-level views because it interacts with the program at an abstract level, by sending and monitoring messages to objects. Because views are coupled to objects' interfaces, rather than implementations, visualisation programmers need only know how to *use* the objects they are visualising, not how to *implement* them, and so do not need to annotate code or analyse data structures within objects' implementations. Because they respect objects' encapsulation, views can be reused to display any object that implements the required interfaces.

2 Aliasing in Object-Oriented Programs

While effective for visualising many smaller programs, this technique is undermined when dealing with larger object-oriented programs. The root of the problem is object identity, one of the foundations of object-oriented programming.

Objects are useful for capturing programmers' intentions to model application domain abstractions precisely because an object's identity always remains the same during the execution of a program — even if an object's state or behaviour changes, the object is always the same object, so it always represents the same phenomenon in the application domain [22,31]. Unfortunately, object identity admits a number of problems due to *aliasing* — a particular object can be referred to by any number of other objects via its identity [19]. The problems arise because objects' states can change while their identities remains the same. A change to an object can therefore affect any number of other objects which refer to it, even though the changed object may have no information about the other objects.

2.1 Aliasing Shadows

Aliases can cause problems for the APMV visualisation scheme whenever a program abstraction is implemented by more than one object in the target program. That is, when there is one *aggregate object* representing the whole of an abstraction, providing an interface to it, and containing one or more other objects implementing the abstraction. We call the objects implementing the aggregate object the members of the aggregate object's *aliasing shadow*[1]. An APMV visualisation depends upon *all* the objects in the shadow, not just the main object providing an interface to the abstraction being displayed.

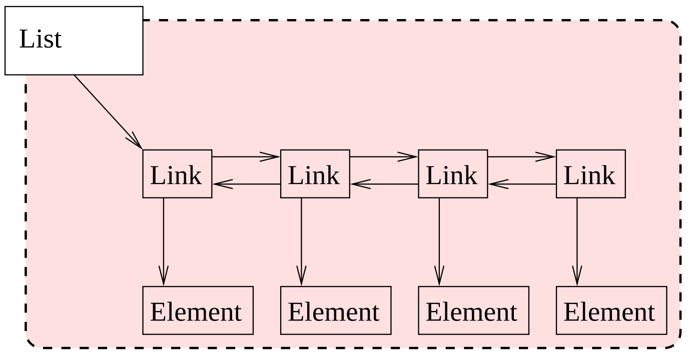


Fig. 5. A linked list and its shadow

For example, consider an object implementing a simple linked list (see Fig. 5). The linked list object is an aggregate, and its shadow contains a doubly-linked list of link objects and the list entry objects actually contained in the list. Although the visualisation system does not access these objects directly (indeed, it may not even know of their existence) the abstract visualisations it produces do depend on these shadow objects.

Aliasing causes problems whenever references cross the boundary of an aggregate object's shadow. Messages can be sent to that shadow object via the alias bypassing the aggregate, and modify the state of the subsidiary objects, and thus of the whole abstraction implemented by the aggregate object. Because these messages were *not* sent to the aggregate object itself — the object actually representing the abstraction — the visualisation system will not detect this message, and so will not be able to produce any change required in the visualisation.

Considering again the linked list, if references to the link objects or the list entries exist outside the linked list collection object (that is, if there are aliases) the contents of the list can be modified by sending a message directly to the link objects, without sending a message to the linked list collection itself.

[1] An aggregate object's shadow is similar to Wills' *demesnes* [48].

2.2 Types of Aliases

Aliasing problems arise in object-orientated programs because the encapsulation boundary enforced in programming languages does not correspond to the abstraction boundary intended by the programmer [18,31]. While an abstract visualisation can be designed to display the *intension* of an abstraction in a program's design, its *extension* as actually implemented in objects in a program can be crucially different.

Aliases can cross abstraction boundaries either inwards or outwards. An inward alias (e.g. a pointer from outside the linked list directly manipulating a link) obviously breaks encapsulation: indeed, such an alias is the symptom of the well-known problem of *representation exposure* — an aggregate's representation is exposed outside the scope of its implementation [23]. Given representation exposure, an external piece of code could change the fields in the link object, easily breaking the invariants of the list and causing the program to crash or loop. Such a change cannot be detected by a visualisation system monitoring the linked list object alone.

Less obviously, outward aliases (e.g. the pointer from the link node objects to the list entries) can also cause problems: this is known as *argument* or *external dependence* [31]. An external alias to the entry could change the entry's value, bypassing the list object itself and thus the visualisation. If external objects' values are part of an aggregate's invariant (say the list is supposed to be sorted) and those values can be changed, then such a change can also break the program.

It is important to realise that not all aliases in object-oriented programs are malign, or at least cause problems for the APMV visualisation model. Rather, only those aliases which cross boundaries into or out of an abstraction's shadow are problematic. An aggregate object (such as the linked list object itself) can be aliased *externally* multiple times — this will not cause any problems for a visualisation of the linked list, provided none of the aliases directly access the links or entries in the list. Similarly, there may be many aliases *internal* to the list aggregate — in a doubly-linked list every link is by definition aliased — but these do not cause problems unless they reach outside the aggregate.

To summarise, we can classify aliases depending upon whether they start and end inside or outside an abstraction, as shown in Fig. 6.

		Source	
		in	out
Destination	in	internal alias	representation exposure
	out	external dependence	external alias

Fig. 6. Categories of Aliases

2.3 Modelling Aliasing

To address these problems, we have developed a model of aliasing in object-oriented programs, based on the idea of object ownership [40,10]. Our model treats object-oriented programs as directed graphs, where objects are the nodes, and interobject references (objects' variables) are the edges. Every object graph has a root node r representing the garbage collection roots — in Smalltalk the object “Smalltalk”, in Eiffel the main class, in C++ and Java the main thread.

Given an object graph, we say one object a owns another object b if and only if every path from the root to b includes a [40]. Ownership gives a simple definition of the interior and exterior of an object. The interior of a is the set of objects a owns — that is, all objects reachable only via a and that would be garbage if a was deleted[2]. The exterior of a is all objects that are not a and are not in the interior of a. Ownership has the important property that if a owns b then an object in the exterior a can never have a reference to b: such a reference would mean that there was a path from the root to b that did not include a.

2.4 Ownership Trees

Ownership is transitive — if a owns b and b owns c then a also owns c and anything else owned by b [40]. Each object has a unique immediate owner, and then a hierarchy of more rarefied transitive owners. For an entire program, every object can be arranged into an *ownership tree* with root r, where every object's parent is its immediate owner.

For example, Fig. 7 shows the ownership tree within two doubly-linked lists that share entries. The root r owns all other vertices trivially since every path from the root includes the root. Each list (a,b) owns their respective link objects (a_n,b_n) because every path from the root to the link objects includes the list object. Since the data c_i is accessible from both lists, it is not owned by either and is promoted to the root. Note that there is not necessarily an edge in the object graph between a parent and a child object in the ownership tree. For example, list a owns the link object a_3 but there is no edge between a and a_3. This definition also accounts for cycles in the graph, such as the double links within the linked listed.

In graph-theoretic terminology objects' owners are known as *articulation points* or *dominators* [2,26]. Dominators and dominator trees are widely used to analyse control flow graphs in compiler theory; Potter, Noble and Clarke [40] recognised the use of ownership to identify structure in programs' object graphs.

2.5 Ownership, Encapsulation, and Aliasing

Ownership is important because it models the extent of encapsulation within object-oriented programs [40]. The key property of ownership is that if a owns b

[2] The interior of an object is similar to the objects in an Island [18], Balloon [1] Confined Type [5], or Ownership Type [10].

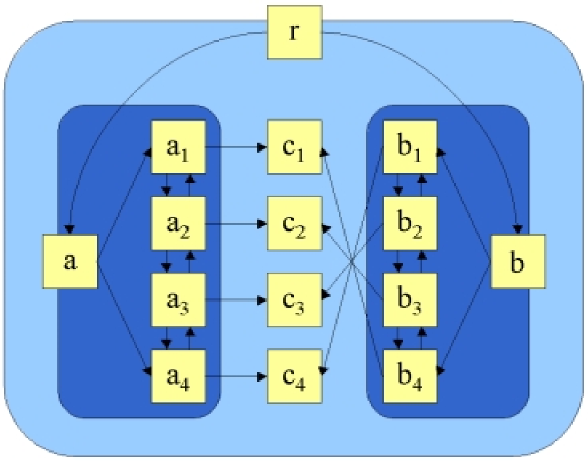


Fig. 7. Two linked lists sharing data

then *all* references to b must pass via a: the corollary to this is that *no* references can reach b from the exterior of a. That is, a encapsulates b, so b cannot be aliased outside a. So long as an object's representation is part of its interior, its representation cannot be exposed.

Ownership also models external dependence. In an ownership tree, an object is directly owned by only one object, so shared external objects must be promoted to more senior positions in the ownership tree — that is, they must be closer to the root than any of the objects that refer to them. The sharing scope of an object is the object's level in the ownership tree. Hence, an ownership tree provides information about the level of aliasing of the objects that it describes. This is why the shared c_i objects are owned by the root in Fig. 7.

2.6 Visualising Aliasing

We have designed and implemented a visualisation of the ownership trees in Java programs [17,16]. Our layout for a single ownership tree rooted at a stack frame is shown in Fig. 8, for the object graph from Fig. 7. This visualisation exhibits a more obvious tree structure and requires significantly less space than the layout in Fig. 7.

In this visualisation, rectangular icons represent objects; each object can be labeled with an individual name or the name of the class to which it belongs. An ownership bar extends horizontally from the top of every aggregate object, that is, every object that owns at least one other object. The interior of an object (that is, all the objects that are encapsulated within that object) is displayed under their owner's ownership bar.

Arrows between vertices represent interobject references. References to a child or an ancestor are denoted by an arrow pointing to or from an owners-

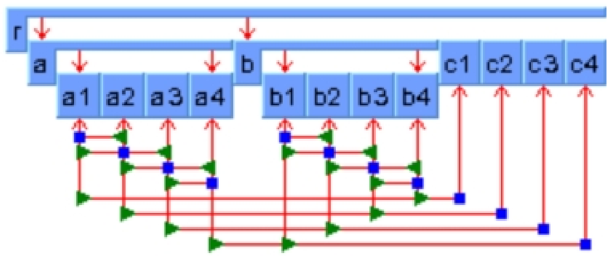


Fig. 8. Ownership Tree Visualisation

hip bar. In the example, r refers to a and b, a refers to a_1 and a_4 and b refers to b_1 and b_4 in this manner. We call these references *inline* references.

References to the child of an ancestor cannot be displayed inline and so extend below the tree. To show such references, we introduce anchor symbols. An anchor for an object a is a global point for references whose destination is a. In the visualisation, an anchor for an object is represented by a small square some distance directly below that object. If an object b refers to a and cannot use an inline reference, the visualisation will show a corresponding arrow from b to the anchor of a by first moving vertically to the height of the anchor, then moving horizontally to the anchor. In the example, a_1 refers to a_2 and c_1 in this manner. We call these references *baseline* references. An anchor is allocated to any object that has an incoming baseline reference from any other object. We also use director symbols — small triangles placed at the intersection of the vertical line from the source and the horizontal line to the anchor. The director helps avoid ambiguity as well as indicating the direction of the anchor.

One of the most important practical advantages of ownership tree visualisations is only shown implicitly in Fig. 8: simply, that the visualisation's layout is a tree. While graphs are notoriously difficult to lay out [3,35,44], tree layouts such as our visualisation can be laid out mechanically. In practice, this means that visualisations can be produced automatically, without requiring any user interaction to place nodes, and can scale up to visualise large systems [15].

Because the ownership tree models encapsulation, the ownership structure below an object cannot be accessed from outside that object. The user can choose to collapse the children of an object, presumably if the interior of that object is not important in the current view of the object graph (see Fig. 9). Any node in the graph can be collapsed: an entire stack, for example, can be collapsed as a unit, hiding all trees rooted in that stack. Because of the property of ownership trees that references cannot break into objects' interiors, collapsing a node can only ever hide that node's internal structure: it can never lose information about outside references. When an object is collapsed, all references to or from a descendent will be visualised as if it were a reference to or from the closest visible ancestor of that child. We call these indirect references, and change the colour of the director. For example, Fig. 9 how linked list nodes can be collapsed in a visualisation. There is an indirect reference between the lists and the element objects they contain, indicating that the lists can access the elements, but do not do so directly.

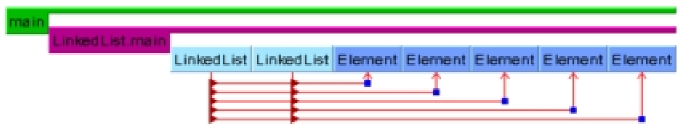


Fig. 9. Two linked lists with internal nodes collapsed

2.7 Alias-Aware Abstract Visualisations

Our ownership tree visualisation is a low-level visualisation that works bottom up, determining and displaying the implicit structure of aliasing in object-oriented programs. Unlike abstract visualisations, ownership tree visualisations do not make any attempt to display programmers' intentions. Also unlike abstract visualisations, ownership tree visualisations do not have any problems with aliasing — rather the reverse: the ownership tree makes any aliasing in the program explicit.

We are developing techniques to use the aliasing information from ownership trees to support abstract visualisations. The key here is the difference between an *aliasing shadow* (the set of objects upon which a visualisation of some aggregate object depends, §2.1) and an aggregate object's *interior* (the set of objects reachable only via that aggregate, §2.3). If an object's shadow is completely contained within that object's interior, then there can be no aliasing problems affecting an abstract visualisation: the only way the object can be changed is via messages sent to its interface.

On the other hand, if an object's shadow is partially or completely *outside* the object's interior, then the object is susceptible to modifications of the external shadow objects[3] via aliases. Once we have identified these objects, we can ensure that they too are monitored by the visualisation system. In this way we can detect changes to aliased objects which may affect the visualisation, and ensure that the visualisation is updated appropriately. We are using currently dynamic analyses to determine objects' interiors and shadow sets [30], however, we are also interested in developing suitable static analyses, such as that for Confined Types [13].

For example, consider for the last time the doubly-linked list in Fig. 10. Both link and list element objects are part of the lists shadow: all these objects are required to produce an abstract visualisation such as that in Fig. 1. The link objects are also part of the linked list's interior, that is, they are not aliased outside the list (there is no representation exposure). In contrast, the elements

[3] We call the part of the shadow in an object's interior the *umbra*, and the part of the shadow outside the object the *penumbra* [30].

are aliased externally, perhaps they are members of other lists (external dependence). In this situation, both the linked list object and the list elements must be monitored to produce a correct abstract visualisation.

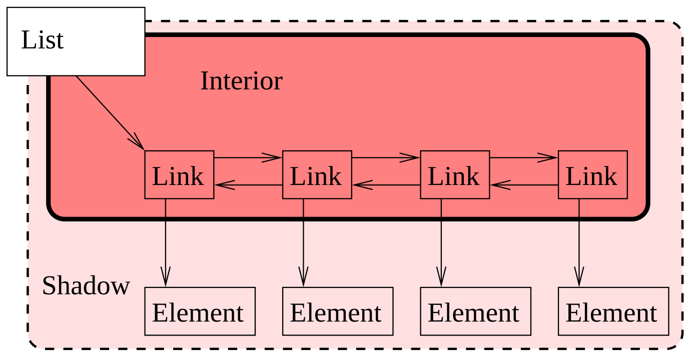


Fig. 10. Shadow + Interior

3 Alternative Approaches

Given the success of object-oriented programming, and the endemic nature of aliasing within object-oriented programs, it is perhaps surprising that more work has not dealt directly with visualising aliasing. It is important to realise the problems we have described in this chapter come from the interplay of aliasing and abstraction: many alternative approaches to visualisation avoid one or both of these concerns.

For example, a large majority of visualisation systems for object-oriented programs produce only low-level views of individual objects — that is, they avoid abstraction. For example, graphical debuggers such as the Smalltalk Program Explorer [4] and DDD [49,50] have gained some practical use, while the IBM program explorer and the Self programming environments were similar research systems [20,21,38,42]. Incense [27] produced displays of records and pointer structures even before object-orientation became widespread.

More recently, several visualisations have been designed to display the behaviour of large numbers of objects simultaneously. Systems such as the Object Visualizer, the Interaction Scenario Visualizer, and J-Insight can display details of object creation and deletion, method invocation, and analyse recurring patterns within programs' execution traces [20,36,37]. The more flexible of these systems require programmers to annotate programs in advance, describing the abstractions they contain and identifying the implementation components whose behaviour should be monitored to produce the visualisations [41,47]. Most of these systems focus on programs' execution behaviour rather than relationships between objects. Notable exceptions include Super-Jinsight, which can display objects referred to or referring to a selected object [36,37,39], and the extended DDD system which uses Sugiyama[4] to display the entire graph [50].

Abstract visualisations are generally the province of algorithm animation systems, such as the paradigmatic BALSA [7], its offspring POLKA [43], and more recent systems such as Pavane [11], Leonardo [12], Jeliot [14], or Jcat [8]. While such systems can produce abstract views, they do so by working bottom up, relying on code annotations or reverse engineering of data structures to build up abstract information out of concrete implementations. Inasmuch as these systems deal with aliasing, they place the responsibility squarely onto the programmer of the visualisations: the annotations or data structure mapping rules must encompass the whole shadow of the aggregate objects to be visualised.

Visualisation systems based on constraint languages such as FORMS/3 [9], Garnet [28], or Animus [6] can also produce abstract visualisations by working bottom-up, however they can often deal with aliasing more easily than visualisation systems for traditional imperative or object-oriented languages. This is because imperative constraint languages automatically accumulate dependencies for constraint expressions so that they can be maintained when input variables change. This means they must compute the shadow of an aggregate object, and will monitor it against changes due to aliasing — even if the shadow is completely encapsulated within the object's interior, when such monitoring is unnecessary. Of course, constraint maintenance imposes a pervasive cost on programs, so constraint languages are not as ubiquitous as object-oriented languages. Constraints can also be used to connect visualisation systems to imperative languages [24]. The Leonardo system for example, runs programs with a sophisticated reversible C interpreter and uses a Prolog-like constraint language to produce graphics [12]. Finally, visualisations can be constructed of programs in functional languages such as Haskell and Miranda. Unlike object-oriented languages, functional languages maintain abstraction but eschew object identity and its concomitant mutable state, and so are effectively immune from these kind of aliasing problems.

4 Conclusion

Object-oriented programming is based on the conceit that objects represent concepts in the "real world". This leads to two fundamental principles: abstraction (objects represent abstract concepts modelled by their interfaces, rather than rather than the code in their implementation) and identity (objects remain individually distinguishable as their states change throughout the program).

The abstraction afforded by objects facilitates abstract visualisation: we can display the concepts represented by objects by monitoring and sending messages via objects' interfaces. Unfortunately, object identity undermines this model: because objects can refer to other objects and object's states can change, aggregate objects can be changed implicitly, without messages crossing their interfaces.

[4] "An algorithm, not a person" [45]

Object ownership can help resolve these problems. By modelling the aliasing structure of an object-oriented program, an ownership tree can identify the implicit encapsulation and containment relationships latent within the program. Visualising an ownership tree can make these implicit relationships apparent. By comparing objects' aliasing shadows and their interiors, we can determine which objects may be affected by aliasing and ensure that changes caused by aliasing are reflected in abstract visualisations.

References

1. Paulo Sérgio Almeida. Baloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
2. S. Alstrup and P. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical report, Dept. of Comp. Sci., Univ. of Copenhagen, 1996.
3. G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Graph drawing, 1999.
4. Kent Beck. Pictures from the object explorer. Technical report, First Class Software, Inc., 1994.
5. Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA Proceedings*, 1999.
6. Alan Borning and Robert Duisberg. Constraint based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4), October 1986.
7. Marc H. Brown. *Algorithm Animation*. ACM Distinguished Dissertation. MIT Press, 1988.
8. Marc H. Brown, Marc A. Najork, and Roope Raisamo. A Java-based implementation of collaborative active textbooks. In *IEEE Symposium on Visual Languages*, pages 372–379, 1997.
9. Paul Carlson and Margaret M. Burnett. Integrating algorithm animation into a declarative visual programming language. In *IEEE Symposium on Visual Languages*, 1995.
10. David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.
11. Kenneth C. Cox and Gruia-Catalin Roman. A characterization of the computational power of rule-based visualization. *Journal of Visual Languages and Computing*, 5(1), March 1994.
12. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, April 2000.
13. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA Proceedings*, 2001.
14. J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *IEEE Symposium on Visual Languages*, pages 360–367, 1997.
15. Trent Hill, James Noble, and John Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS Pacific 2000*, Sydney, 2000. IEEE CS Press.
16. Trent Hill, James Noble, and John Potter. Visualising the structure of object-oriented systems. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 191–198, Seattle, 2000. IEEE CS Press.
17. Trent Hill, James Noble, and John Potter. Scalable visualisations of object-oriented systems. *Journal of Visual Languages and Computing*, 2002. To appear.

18. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.
19. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.
20. Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA Proceedings*, October 1995.
21. Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5), May 1997.
22. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
23. Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.
24. John H. Maloney, Alan Borning, and Bjorn N. Freeman-Benson. Constraint technology for user-interface construction in ThingLab II. In *OOPSLA Proceedings*, 1989.
25. Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and Akinori Yonezawa. A general framework for bi-directional translation between abstract and pictorial data. *ACM Transactions on Information Systems*, 10(4), October 1992.
26. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.
27. Brad A. Myers. Incense: A system for displaying data structures. In *SIGGRAPH Proceedings*, 1983.
28. Brad A. Myers, Rich McDaniel, Rob Miller, Brad Vander Zanden, Dario Guise, David Kosbie, and Andrew Mickish. The prototype-instance object systems in Amulet and Garnet. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 7. Springer-Verlag, 1999.
29. James Noble. Prototype-based programming for abstract program visualisation. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Conecepts, Languages, Applications*. Springer-Verlag, 1997.
30. James Noble and John Potter. Change detection for aggregate objects with aliasing. In *Australian Software Engineering Conference (ASWEC)*, 1997.
31. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.
32. R. James Noble. *Abstract Program Visualisation: Object Orientation in the Tarraingím Program Exploratorium*. PhD thesis, Victoria University of Wellington, December 1996.
33. R. James Noble and Lindsay J. Groves. Tarraingím — a program animation environment. *New Zealand Journal of Computing*, 4(1), December 1992.
34. R. James Noble, Lindsay J. Groves, and Robert L. Biddle. Object oriented program visualisation in Tarraingím. *Australian Computer Journal*, 27(4), November 1995.
35. Stephen North. Visualizing graph models of software. In John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualisation*. M.I.T. Press, 1998.
36. W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *COOTS Proceedings*, 1998.
37. W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *OOPSLA Proceedings*, 1999.

38. Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA Proceedings*, October 1993.
39. Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the Execution of Java Programs. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
40. John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference (ASWEC)*, 1998.
41. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-oriented visualization. In *OOPSLA Proceedings*, 1996.
42. Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *OOPSLA Proceedings*, 1995.
43. John T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3), September 1990.
44. John T. Stasko and Charles Patterson. Understanding and characterising software visualisation systems. In *IEEE Workshop on Visual Languages*, 1992.
45. Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man and Cybernetics*, 11(2):109–125, 1981.
46. David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp And Symbolic Computation*, 4(3), June 1991.
47. R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA Proceedings*, 1998.
48. Alan Cameron Wills. *Formal Methods applied to Object-Oriented Programming.* PhD thesis, University of Manchester, 1992.
49. A. Zeller and D. Lütkehaus. DDD — a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1), 1996.
50. Thomas Zimmermann and Andreas Zeller. Visualizing Memory Graphs. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*

Algorithm Animation Using Data Flow Tracing

Jarosław Francik

Silesian University of Technology,
Institute of Computer Science,
ul. Akademicka 16, 44100 Gliwice, Poland
jfrancik@polsl.gliwice.pl

Abstract.

Successful algorithm visualization inherently provides a high level of abstraction, supplying an extra information on the semantics that is behind the code. The postulate of designing highly abstract visualizations is stated as being in a deep contradiction with postulate of automation of the designer's work. The goal of the presented work is partial reconciling these two contradicted postulates. Some elements that significantly increase the level of abstraction may be introduced to the visualization in a strictly automatic mode. To obtain this result, an original method of algorithm animation based on data flow tracing is proposed. Its key idea is to acquire information by observing elementary operations of data flow. For dynamic analysis of non-local flows Petri net formalism is used. The new method has been successfully applied in an algorithm animation system Daphnis.

1 Introduction

The comprehension of the way, in which algorithms work, is one of the most significant requests pointed before software designers. It is an essential part of both education and engineering practice in computer science.

The way in which an algorithm is written has a great influence to the degree how it can be understood. The notation in a formal programming language is most precise, but, in the same time, worse adapted to the human system of perception [5]. It does not support the process of comprehension of an algorithm by a human being. A description in a natural language is not much better: both methods of description are textual; they present the described object analytically, step by step, but to support the synthetic sight, essential for the comprehension of the problem, it is necessary to get a pictorial description, which is much closer to the human perception model.

Software visualization [21] is a technique that supports human understanding of computer software applying various, but primarily pictorial, means of expression. These means are what we call multimedia. If the subject of such visualization is an algorithm, and computer generated animation is essential for its realization, we say of algorithm animation.

The quality of a visualization – readability, comprehensibility, clearness – is influenced by several factors. One of them is the capability of creating projections at a high level of *abstraction* [19][22]. It is crucial for human understanding of the software. Visualizations that provide high level of abstraction go beyond isomorphic mapping data structures (or code) to graphics. They supply an extra information on the semantics that is *behind* the code, present neither in the executed program nor in its data; it is user-conceptualized information of what is important. This information, called the *intention content* [22], supports human understanding of the software.

Another factor, that is not so important for the observer, but it is crucial for the user who creates a visualization (the visualizer), is the level of *assistance* and *automation* offered by the system. It is provided in the visualization systems in various form: from manual preparation mode with little or no assistance to solutions that are fully automated.

By most authors the postulate of providing high level of automation is stated as deeply contradictory with the postulate of obtaining high level of abstraction [17][18][19]: the more abstract a visualization should be, the less use of automatic facilities may be achieved to create it, and vice versa.

The method proposed in this paper is a means for partial reconciling the two contradicted postulates. It does not make visualization preparation fully automatic: it just introduces some elements that make the level of abstraction higher without any additional effort of the visualizer.

2 Related Work

Many efforts have been done to classify software visualization tools. One of the first attempts was a six-category taxonomy made by Myers [14]. He developed his classification using two axes: the level of abstraction (whether the systems illustrate the code, data or algorithm of a program), and the level of animation in their displays (whether they are static or dynamic). This resulted in a two by three grid. Stasko and Patterson [22] introduced scaled dimensions in their four-category taxonomy covering aspect, abstraction, animation and automation. Price, Small and Baecker have proposed the most detailed and comprehensive taxonomy [17]. Their classification involves 47 categories, grouped in six main super-categories: scope, content, form, method, interaction and effectiveness.

One may notice that in most classification systems abstraction plays important part. Let us review software visualization packages with focus on their level of abstraction in conjunction with their degree of automation.

A pioneering Brown and Sedgewick's *BALSA* system [2] (with its successor, *Zeus* [1]) has become a benchmark against which all subsequent systems have been measured. The level of abstraction is high, the visualizer may generate a great deal of the intention content, but the mode of preparation is basically manual, without any strong automated tools. Stasko's *Tango* and *XTango* [20][23] systems, as well as their successors like *Polka*, may provide even higher level of automation with relatively less effort of the visualizer. An external specification of visualization has been applied, that aids generating abstractions: the animation effects that represent algorithm's operations. Thanks to an original *path-transition paradigm* [24], providing smooth animation is relatively easy. The designer's activity is more comfortable, but still all the abstraction concepts have to be programmed manually. The *Animus* system (by London & Duisberg [4]) is a mile stone in the history of visualization systems thanks for its declarative form of visualization specification, that does not involve any modification in the source text. The level of abstraction obtained is however low and there are no automatic facilities that would support it. Another declarative system is *Pavane* [12][16]. It provides tools for generating high-level abstractions; they are not automatic, but they are powerful and in the same time relatively easy-to-use. Another example of a declarative animation system is Leonardo [3].

On the other end there is a number of systems that provide partial or full automation, but they produce simple visualizations, with low level of abstraction. In the *Siamoa* system [9] simple visualizations may be produced in a partially automated way; they provide low level of abstraction. It is also possible to achieve higher level of abstraction, but it involves handcrafted, manual design. To this group belongs also an early *Incense* system created by Myers [15]. Its modern successors may be found in commercial software developers' packages, in form of semi-graphical watch windows and source text browsers.

An interesting middle-point is *UWPI* system [13], that could automatically visualize abstract data structures. It contained a set of schemas and rules to be applied to the actual data structures. A special *inferencer* unit recognized and analyzed data structures and proposed plausible abstractions, depending on set of operations detected during this analysis. Another unit, *layout strategist*, created final graphical representation. The system was not competitive to the leading solutions in the field of abstraction, but the level of automation was impressive: it dramatically outstands other automatic systems. The system is not developed further because of financial reasons.

This short review seems to confirm the general contradiction between automation and abstraction (maybe with exception of *UWPI*). What lacks the software visualization systems, is the ability to create abstractions automatically. Price, Baecker and Small made of this ability one of the 47 categories in their taxonomy, calling it **intelligence**.

3 Three Postulates of Successful Algorithm Animation

Looking for a model of algorithm animation that involves some basic level of abstraction, a principal postulate of successful visualization has been formulated: illustrating the data flows. Further research led to another two postulates. All these three postulates correspond to the aspects of abstraction that have been stated previously. Below they are discussed in detail.

Postulate of illustrating data flows: visualization should apply smooth animation to render operations of data flow, which occur during the process of algorithm execution. A good visualization has to illustrate not only the states a program reaches during execution but also how the transformation between the states occurs. A way to obtain this is a visualization in which the graphical representation of data is smoothly animated from the place representing its source location towards the place representing its destination. It passes through a series of in-between positions that have no correspondence to the actual state of neither the program nor its data. Therefore the graphical objects do not represent variables, but rather its persistent contents flowing from one location to another.

Postulate of spatial suppression of information: visualization should suppress information concerning some variables considered to be unimportant. A visualization that conforms this postulate has for its subject an implementation of the algorithm different then the actual one, as it contains less number of variables (and data flow operations). Suppressed are those elements, which do not support understanding of the algorithm. Ignoring some variables would often lead to ceasing the coherence of the global image of data flow in the program. Often variables supposed to be unimportant make an intermediate stage of a wider data flow that begins and ends inside the user's area of interests. It involves the need of reconstruction of broken multi-stage data flows. Such data flows should be glued so that, if possible, long-distance, single flows are formed, in which both source and destination variables are important – from the user's point of view.

Postulate of temporal suppression of information: a part of information concerning ordering in time of elementary data flow operations should be removed from the projection. Especially, some operations that are actually executed sequentially may be rendered as if they were executed in parallel and synchronously. The sequential program is an overspecification of the algorithm. Thus, the rendered realization of an algorithm differs from the one actually applied in the visualized piece of software. The detailed information about time ordering of operations, that are independent and executed in short period, often not only gives no support for algorithm understanding, but – on the contrary – it draws the observer's attention away from other, much more important elements of the projection.

The above three postulates may be illustrated using an example of the operation of exchange of two values (fig. 1). A simple, static visualization shows the state before and after the operation, giving no directions on how it was performed (fig. 1a). An animation that uses first postulate (illustrating data flows) gives information on what happens: what is source and destination of data. An animation made in accordance with all three postulates (fig. 1c) hides unimportant details (an auxiliary variable, order of assignments) and shows what is most important: an exchange of two values as it was a synchronous, symmetric process. It is closest to intuitive meaning of an "exchange", with all the implementation details hidden.

Satisfying all the three paradigms involves some level of abstraction. Most existing visualizations go in this direction; good examples may be numerous animations of sorting algorithms: elementary exchange operations almost always are illustrated as symmetric and synchronous processes of moving graphical objects, without use of any auxiliary variables. In the systems existing so far, obtaining such a visualization involves an a priori, handcrafted design. The proposed method for algorithm animation, described below, allows obtaining such visualizations automatically.

4 Data Flow Driven Method of Algorithm Animation

The method of algorithm animation based on data flow tracing allows obtaining visualization that satisfy all the three postulates mentioned in previous section in a way strictly automatic. In general, it is based on some instruction scheduling features. This method is an essential part of the research presented in this paper.

4.1 Data Flow Driven Animation Engine

The process of algorithm execution or, to be more precise, of execution of a program that realizes an algorithm, may be effectively described as a sequence of data flows. This should not be confused with a description of an algorithm itself; this model tells

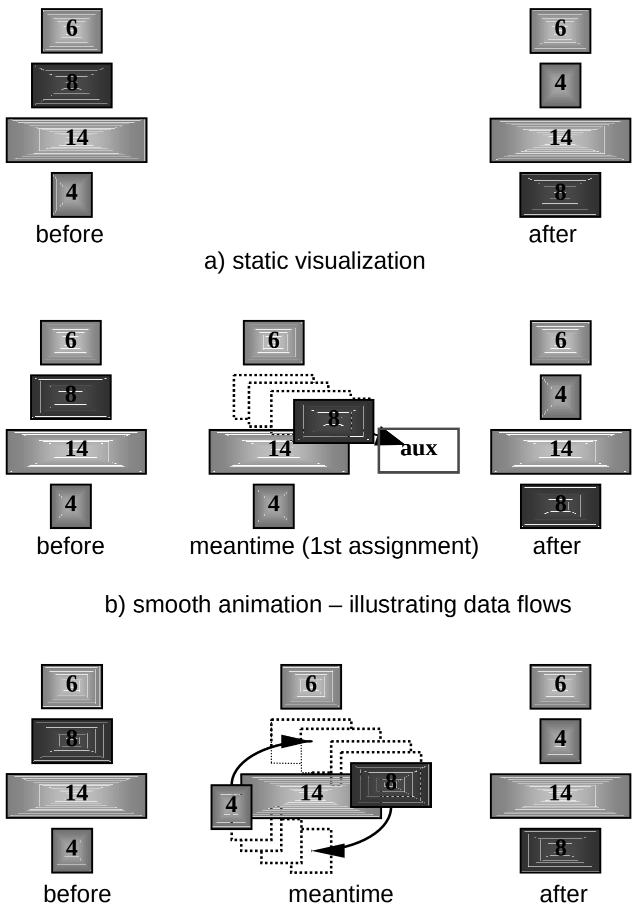


Fig. 1. Modes of visualization of exchange of two values

only what happens during a program execution and may not be used as a tool for algorithm specification. The crucial terms are:

Variable – an entity with attributed name and type, and *optionally* a value. A variable is called *valuated* if it has a value.

Valuating a variable – giving a variable a (new) value. Normally variables have no values before they are valuated for the first time (exception: variables keeping input parameters).

Data flow – a phenomenon that causes valuating a destination variable with a value defined by a set of zero or more source variables. During a typical program execution the set of valuated variables extends while consecutive data flows occur.

Once the process of algorithm execution may be described in terms of data flows, it is possible to outline a data-flow-driven algorithm animation engine. This engine satisfies the postulate of illustrating data flows.

Traditional debugging and visualization tools obtain data from the observed program in shape of a stream of information that concerns consecutive changes of values of the observed variables. In the proposed engine, this is rather a series of information on elementary data flow operations that form the input data stream for the system.

The animation is kept in movement by a series of consecutive calls to routines reporting elementary data flow operations. Before any variable is visualized, it has to be registered to the system, so that it may be identified and inspected. When the visualization ends, the variables should be unregistered. This has to occur before the variables cease to exist. This constitutes the three basic function calls to the engine, called *Play*, *Register* and *Unregister*. They make the only functions, with which a source text of program being visualized has to be annotated.

To prepare a visualization, it is necessary to supply an external configuration script, which specifies graphical representation and rules of translation for all the variables to be visualized.

The attributes of the image may depend on the value of a variable or on its address. During the data flow driven visualization, the graphical elements are connected rather with values stored in variables, not with variables. During a typical data flow operation, this is not only the value, that changes, but also the address of this value. In the discussed engine values are represented by size and shape of corresponding graphical objects, while their addresses are represented by placement on the screen. Thus, change of value of a variable is rendered as a deformation of its graphical representation, while change of address of a value (a data flow operation) is depicted as a movement of a graphical object from the position representing the source address to the position representing the destination address. In simplification, the value may be connected with ‘width’ or ‘height’ attributes of a graphical object, while its address is connected with ‘x’ and ‘y’ attributes.

4.2 Petri Nets Applied to Describe Algorithm Behavior

The engine proposed in previous subsection meets the first of the three postulates of successful algorithm animation: illustrating data flows. To meet the two other postulates, i.e. spatial and temporal information suppression, the Petri net formalism has to be applied to describe the process of algorithm execution.

An algorithm execution process may be described with a Petri net in which places represent variables, and transitions represent elementary operations of data flow. An example is shown in fig. 2.

The sequence of data flow operations in an algorithm is strictly determined. Thus also firing transitions should be strictly determined. However, in typical Petri nets time of firing is not determined. A model based on such nets does not allow to define sequences of operations executed in order. What's more, in each enabled transition may be potentially fired, what is obviously not the situation that we meet in algorithms. One of possible solutions would be to apply non-autonomous net, or timed Petri nets.

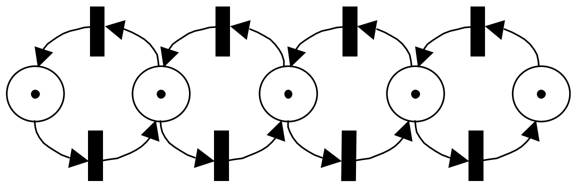


Fig. 2. Bubble sorting described using a Petri net

4.3 Step of Algorithm and Its Features

The approach presented below uses a regular, autonomous Petri net to describe the algorithm's behavior. To avoid problems connected with undetermined nature of such nets, the description has to be limited to a fragment of the algorithm execution time that is short enough to make the order of execution unimportant. This short fragment is called a *step of algorithm*.

Definition 4.1. A step of algorithm is such a set of executed consecutively operations, that it is possible to execute them in any order without changing the meaning of the algorithm.

Definition 4.2. A synchronous step of algorithm is such a set of executed consecutively operations, that it is possible to execute them in parallel, synchronously, without changing the meaning of the algorithm.

Example:

```
1: a = 5;
2: b = 10;      // step #1
3: c = a + b;
4: d = a * b;   // step #2
5: c = c + d;   // step #3
```

One can easily notice, that instructions *1* and *2* may be executed in any order or in parallel without change of the meaning of the algorithm; the same situation with instructions *3* and *4*; but none of the instructions *3* and *4* could be executed before neither *1* nor *2*. Also instruction *5* must be executed after instructions *3* and *4* are finished.

A sequence of operations that constitute an algorithm, may be divided into a series of successive steps. To draw the rules of such division a few more definitions should be introduced.

Let $PN(P, T, F)$ be a Petri net that models some sequence of operations executed in succession. P is set of places, T is set of transitions and F is the flow relation of this net. Let a function $\tau(t)$ designates the time of firing the transition t. Let $*t$ be the input set of a transition t, and $t*$ be the output set of a transition t.

Definition 4.3. Operations represented with transitions t_1, $t_2 \in T$, where $\tau(t_1) < \tau(t_2)$, are called **dependent**, if $t_1* \cap *t_2 \neq \varnothing$. We say that there is a relation of *dependency* between them (fig. 3a).

Definition 4.4. Operations represented with transitions t_1, $t_2 \in T$ are called *sequential*, if $t_1* \cap t_2* \neq \varnothing$. We say that there is a relation of *sequence* between them (rys. 3b).

Definition 4.5. A single-step net is a Petri net, in which all the represented operations belong to the same synchronous step of algorithm.

Now it is possible to formulate the rule, on basis of which the general algorithm is built:
Lemma 4.1. The Petri net *PN(P, T, F)* is a single step net when and only when:

$$\forall t_1, t_2 \in T: \quad \tau(t_1) < \tau(t_2) \Rightarrow t_1{*} \cap t_2{*} = t_1{*} \cap {*}t_2 = \varnothing$$

In other words, any two operations belong to the same synchronous step of an algorithm when and only when they are neither dependent nor sequential.

The proof of this lemma may be found in [6][7][8].

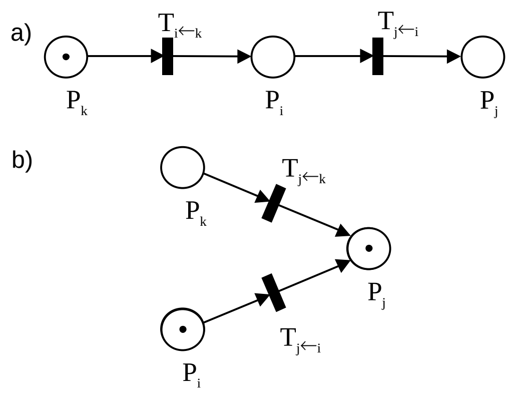


Fig. 3. Cases, in which transition *T* may not be fired in parallel: a) dependency; b) sequence

4.4 Single-Step, Single-Color Algorithm

The single-step, single-color algorithm (table 1) is executed each time when an elementary data flow operation is reported to the visualization engine. It corresponds to the operation *Play* as it was defined in section 4.1.

The key idea of the algorithm is to collect consecutive operations that belong to the same synchronous step of the algorithm. If a new reported operation exceeds the current step, all the operations collected so far are animated in parallel, synchronous mode. Thus, the whole animation is divided into parts that correspond to the consecutive algorithm steps, and operations that belong to each step are presented synchronously.

To determine if a reported operation belongs to the current step or exceeds it, the lemma 4.1 is applied. It means that the operation is checked if it is dependent or sequential.

The presented algorithm satisfies the postulate of temporal suppression of information, but does not yet satisfy the postulate of spatial suppression of information. It constitutes a first approach to the actual solution.

Table 1. Single-step, single-color algorithm

```
// PN is a Petri net where operations that
// belong to the current step are collected
Play (Fj←i) // reporting Fj←i data flow operation
{
    // OP is Petri net representation of Fj←i
    OP = {pi → tj←i → pj};
    bool dependency = false;
    bool sequence = false;

    // test for dependency and sequence
    for (each transition ty←x ∈ PN)
    {
        if (i == y)  dependency = true;
        if (j == y)  sequence = true;
    }

    if (dependency || sequence)
    {
        Animate(PN);
        PN = ∅;
    }
    PN = PN ∪ OP;
}
```

4.5 Spatial Transformation of a Synchronous Step

In order to satisfy the postulate of spatial suppression, appropriate transformations of the single-step Petri net have to be introduced.
Definition 4.6. Let *PN(P, N, F)* be a net and let:

$$\{ p_i \rightarrow t_{x\leftarrow i} \rightarrow p_x \} \cup \{ p_x \rightarrow t_{j\leftarrow x} \rightarrow p_j \} \in PN$$

A transformation of *gluing* $P_i - P_j$ around P_x we call such a transformation of the net *PN*, in result of which a net *PN'* is created, in which operation $\{ p_x \rightarrow t_{j\leftarrow x} \rightarrow p_j \}$ is replaced with $\{ p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j \}$ (fig. 4a):

$$PN' = PN - \{ px \rightarrow tj\leftarrow x \rightarrow pj \} \{ pi \rightarrow tj\leftarrow i \rightarrow pj \}$$

Definition 4.7. Let *PN(P, N, F)* be a net and let:

$$\{ pi \rightarrow tx\leftarrow i \rightarrow px \} \{ pk \rightarrow tx\leftarrow k \rightarrow px\} PN.$$

A transformation of *interception* of $P_i - P_x$ by P_k we call such a transformation of the net *PN*, in result of which a net *PN'* is created, in which operation $\{ p_i \rightarrow t_{x\leftarrow i} \rightarrow p_x \}$ is removed (fig. 4b):

$$PN' = PN - \{ pi \rightarrow tx\leftarrow i \rightarrow px \}$$

In above definitions $\{ p_x \rightarrow t_{y\leftarrow x} \rightarrow p_y \}$ designates a subset of Petri net which represents two variables and a data flow occurring between them.

In order to formulate the ultimate algorithm for algorithm animation, the following lemma has to be formulated:

Lemma 4.2. The transformations of gluing $P_i - P_j$ around P_x and interception $P_i - P_x$ by P_k do not change the semantics of the algorithm represented by the transformed Petri net, under the condition, that – in both cases – P_x represents an unimportant variable.

The proof of this lemma may be found in [6][7][8].

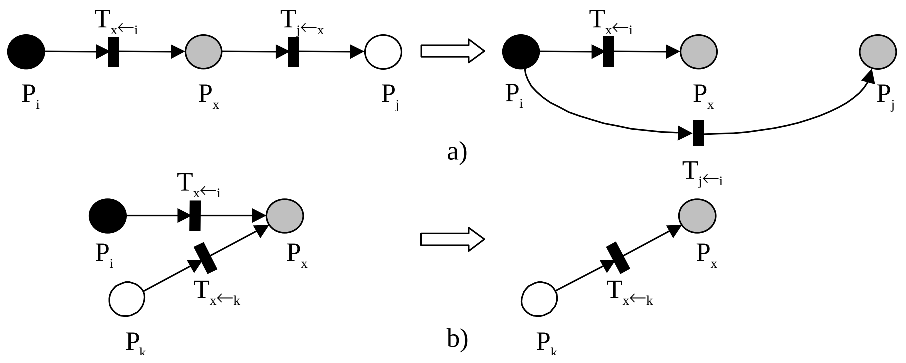


Fig. 4. Application of gluing (a) and interception (b) spatial transformations of Petri nets in single-step, three-color algorithm

4.6 Single-Step, Three-Color Algorithm

The single-step, three-color algorithm (table 2) is an enhancement of the single-step, single-color one. Its goal is to satisfy the postulate of spatial suppression of information. The key idea is to apply spatial transformations of gluing and interception to suppress information about the variables considered to be unimportant. This method guarantees that the coherence of the global image of data flow in the algorithm is kept.

First, all the variables (places) are colored. Places that represent important variables are painted black. If a place represents an unimportant variable, but it is a destination of any data flow operation that originates from a black or gray place, it is painted gray. All the rest of unimportant variables are painted white.

The suppression of information occurs in following two situations:

If a chain of data flow originates from a black place, goes through a gray place(s) and terminates at a black place, then the transformation of gluing may be applied to suppress the information about the gray variable(s). Obviously it is enough to detect if the source place of a data flow operation is gray, and if it is, to apply the gluing transformation (fig. 4a).

If some data flow operations share the same destination place, and this place is not black, then the transformation of interception may be applied so that to keep only the information concerning the latest data flow (fig. 4b). Such sequential assignments could be meaningful if they shared a black place as their destination (it might be a cycle or just a dead code), but otherwise they have no influence for any further processing involving important variables.

In both situations, according to the lemma 4.2, the transformations applied do not change the meaning of the algorithm, and in the same time they suppress unwanted spatial information. Thus, the algorithm satisfies all the three postulates of successful animation.

Further improvements of this algorithm lead to 1,5-step, three colored algorithm, in which some data flow chains may be glued across the limits of consecutive synchronous steps of algorithm. The final tuning involved also some detailed limitations in collecting operations, especially when visualizing iterations.

Table 2. Single-step, three-color algorithm

```
Play ($F_{j\leftarrow i}$)
{
     OP = {$p_i \rightarrow t_{j\leftarrow i} \rightarrow p_j$};
     bool dependency = false;
     bool sequence = false;

     for (each transition $t_{y\leftarrow x} \in$ PN)
     {
         if (i == y)
         {
            d = x; dependency = true;
         }
         if (j == y)
         {
            s = x; sequence = true;
         }
     }

     // setting the colours
     if ($p_i$ is in the observer's area
                          of interest)
         $C_i$ = BLACK;
     else if (dependency)
         $C_i$ = GREY;
     else
         $C_i$ = WHITE;

     if ($p_j$ is in the observer's
                  area of interest)
         $C_j$ = BLACK;
     else if (sequence)
         $C_j$ = GREY;
     else
         $C_j$ = WHITE;

     // gluing $p_d$ - $p_j$ around $p_i$
     if ($C_i$ == GREY)
     {
         OP = {$p_d \rightarrow t_{j\leftarrow d} \rightarrow p_j$};
         $C_i$ = BLACK;
         dependency = false;
     }

     // interception $p_s$ - $p_j$ by $p_i$
     if ($C_j$ == GREY)
     {
         PN = PN - {$p_s \rightarrow t_{j\leftarrow s} \rightarrow p_j$};
         sequence = false;
     }
     if ($C_i$ == WHITE
      && $C_j$ != BLACK) return;

     if (dependency || sequence)
     {
         Animate(PN);
         PN = $\emptyset$;
     }
     PN = PN $\cup$ OP;
}
```

5 The Daphnis System

The practical effect of the presented research was development of the algorithm animation system *Daphnis*. The system applies 1,5-step, three-color algorithm described in previous section.

Development works on *Daphnis* system started in 1997. It is a successor of previous systems, *SANAL* and *WinSANAL* [10][11]. An important inspiration was also a prototype system *Siamoa*, created in University of Lille 1 [9]. *Daphnis* distinguishes from its predecessor with its rebuilt internal architecture. The most important innovation is applying the method of algorithm animation based on data-flow tracing.

Daphnis is a general purpose algorithm animation system. It means that it is capable to visualize action of algorithms of any class. The system works under *Windows 9x*, *Windows NT 4.0/2000* or newer operating systems, on *PC* computers. Hardware requirements are like of adequate operating system.

Daphnis system is generally independent of the language, in which visualized algorithm is formulated. The program being visualized has to be a Windows 32 platform module, capable to access dynamic link libraries (*dll*). An application programmer's interface has been created that makes the system's projection engine accessible for programs written in C and C++, so preparing programs for projection is especially easy if they are written in one of these languages. Using any other language requires applying adequate for this language techniques of interface declaration.

Daphnis is a data-driven system, with declarative method of specification of visualization. Some annotations of the source text of programs are still necessary, so style of specification is invasive.

To create a projection in Daphnis three steps must be done:

Annotating the program source text. It runs according to fixed rules, and could be easily automated (works upon an automatic source text preprocessor are in progress). Functions that are to be inserted into the text of visualised program may be divided into two groups. Driving functions, that keep projection in move, is one of these groups. They are essential for the applied method of animation, and were discussed in Part 4. The other group is group of technical (auxiliary) functions, that allow to tune the projection. Some of them duplicates the functionality of the user interface.

Compiling and linking. Functions used to annotate source programs are available in the form of a dynamic link library. It makes them language-independent. Proper header files for C and C++ are also provided. Compiling and linking of C/C++ programs is easy, and preparing programs in other languages is only a bit more complicated. In both cases the result is a ready-to-run, modified executable. It may be run with various configuration scripts, and changing configuration script does not require recompiling.

Providing a configuration script. This is the most difficult stage in projection preparation. To create a script, users not only have to know a language of the scripts, but also know the general rules of projection engine structure. In general, the configuration script contains a series of definitions of objects, called actors. Each actor is connected with an individual variable subjected to visualisation and with graphical element (called grel), that represents the variable. It also contains a set of translation rules, that determine the way, in which values or addresses of the visualised variables should be translated to values controlling various attributes of the final image. Description of the syntax and semantics of the configuration scripts is illustrated by several examples – both simple and more advanced.

6 A Sample Animation

A sample animation of the quick-sort algorithm created with Daphnis is shown in fig. 5. It involved creating a configuration script that defined how the sorted elements should be colored and specified other important variables. The movement of the sorted elements is managed automatically by the system. The annotated source text of this animation, as well as the script file are given below; the bolded statements are the *Daphnis* system annotations. They are numerous, but they are quite easy to use (just three types of system calls have been used in this sample). The system for automatic program annotation is currently also available.

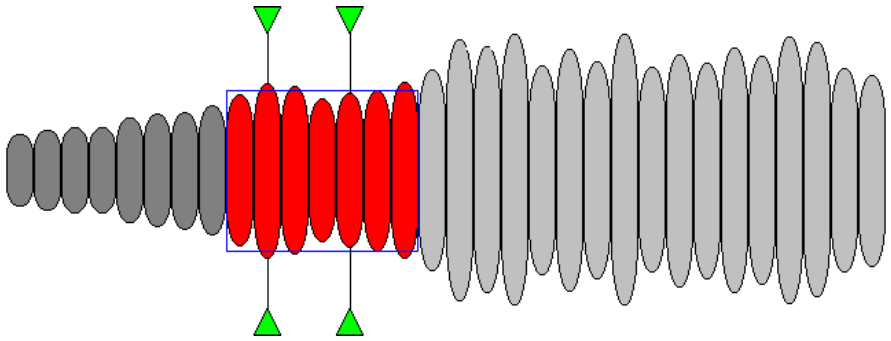


Fig. 5. Quick-sort animation in the *Daphnis* system

```
void Quicksort(int arr[], int p, int r)
{
       if (p < r)
       {      int q = Partition(arr, p, r);
              Quicksort(arr, p, q);
              Quicksort(arr, q + 1, r);
       }
}

int Partition(int arr[], int p, int r)
{                                    Register("p", &p); Play(&p);
                                     Register("r", &r); Play(&r);
       int x = arr[p];               Register("x", &x); Play(&x);
       int i = p - 1;                Register("i", &i); Play(&i);
       int j = r + 1;                Register("j", &j); Play(&j);
       while(1)
       {      do { j--;                     Play(&j); }
              while (arr[j] > x);
              do { i++;                     Play(&i); }
              while (arr[i] < x);
              if (i < j)
              {      int tmp = arr[i];      Register("tmp", &tmp);
                                            Play(&tmp, &arr[i]);
                     arr[i] = arr[j];       Play(&arr[i], &arr[j]);
                     arr[j] = tmp;          Play(&arr[j], &tmp);
                                            UnRegister(&tmp);
              }
              else
              {                      UnRegister(&p); UnRegister(&r);
                                     UnRegister(&x); UnRegister(&i);
                                     UnRegister(&j);
                     return j;
              }
       }
}
```

The quicksort animation script file is also presented. *Daphnis* script files contain relatively simple definitions of graphical objects that may be observed on the screen during the projection. The names of the objects (e.g. *Arr*, *i*, *j*) correspond with the names used in *Register* function calls in the listing above; this is how a link between a graphical definition and an actual value is created. There are some variables registered in the program, and in the same time absent in the script; these are unimportant variables, which lack the graphical representation.

```
Arr is roundRectangle
{      index x = linear(0, 1, 30, 50);
       y = 120;
       width = 20;
       height = linear(0, 100, 50, 200);
       // we use simple calculations to make colors more interesting
       redBrush = (#index < p) ? 128 : (index <= r) ? 255 : 192;
       greenBrush = (#index < p) ? 128 : (index <= r) ? 0 : 192;
       blueBrush = (#index < p) ? 128 : (index <= r) ? 0 : 192;
};
i is arrayPointer
{      value x = linear(0, 1, 16, 44);
       y = 120;
       width = 20; height = 240;
       greenBrush = 255;
}
j is arrayPointer
{      value x = linear(0, 1, 16, 44);
       y = 120;
       width = 20; height = 240;
       greenBrush = 255;
}
x is frame
{      value x = linear(0, 1, 16, 44) of (#p + #r) / 2;
       y = 120;
       width = linear(0, 1, 16, 44) of (#r - #p);
       height = linear(0, 100, 50, 200);
       blueBrush = 255;
};
```

7 Conclusion

What lacks the software visualization systems is the intelligence, or the ability to create abstractions automatically. The presented work is a step towards intelligent systems: some aspects of abstraction in visualization, even if not very strong, may be obtained in strictly automatic mode, thanks to an original method of data flow tracing. This level of abstraction usually involves handcrafted design.

The proposed method has been successfully verified in a practical implementation of the algorithm animation system *Daphnis*. Experiences with use of this system collected so far fully confirm the usefulness of the data flow tracing approach to algorithm animation.

References

[1] M. H. Brown, “Zeus: A System for Algorithm Animation and Multi-View Editing”, *Proc. Of IEEE Workshop on Visual Languages*. Kobe, Japon, 1991, pp. 4-9.

[2] M. H. Brown, “Exploring Algorithms Using Balsa II”, *IEEE Computer*, Vol. 21, N°5, 1988, pp. 14-36.

[3] P. Crescenzi, C. Demetrescu, I. Finocchi, R. Petreschi, “Reversible execution and visualization of programs with LEONARDO”, *Journal of Visual Languages and Computing*, 11(2):125-150, 2000.

[4] R. A. Duisberg, “Animated Graphical Interfaces Using Temporal Constraints”, *Proc. of the ACM SIGGHI'86 Conf. Human Factors in Computing Systems*, Boston, MA, 1986, pp.131-136.

[5] V. Fix, S. Wiedenbeck, J. Scholtz, “Mental Representations of Programs by Novices and Experts”, *Proc. of Inter CHI'93*, Addison-Wesley, 1993, pp. 74-79.

[6] J. Francik, *Surveillance du flux des données dans l’animation des algorithmes*. PhD thesis, University of Lille, Villeneuve d’Ascq, France, 1999.

[7] J. Francik, *Sledzenie przeplywu danych dla potrzeb animacji algorytmów (Data Flow Tracing for Algorithm Animation)*. PhD thesis, Technical University of Silesia, Gliwice, Poland, 1999.

[8] J. Francik, “Algorithm Animation Pages”, http://www-zo.iinf.polsl.gliwice.pl/~jfrancik/aa

[9] J. Francik, P. Szmal, F. Van de Veire, “Siamoa – A System for Visual Programming, Program Visualization and Debugging”, *Proc. of Advanced Visual Interfaces AVI ’98*, ACM Press, L’Aquila, Italy, 1998, pp. 289-291.

[10] J. Francik, P. Szmal, “Algorithm Animation and Debugging with the WinSanal System”, *Proc. of IASTED Conference Applied Informatics*, Innsbruck, Austria, 1997.

[11] J. Francik, P. Szmal, “WinSanal: System animacji algorytmów dla potrzeb dydaktyki i inzynierii programowania”, *Informatyka* 7-8/1997.

[12] D. Hart, E. Kraemer, G.-C. Roman, “*Interactive Visual Exploration of Distributed Computations*”, Washington University, Department of Computer Science, St. Louis, MO, 1997.

[13] R.R. Henry, K.M. Whaley, B. Forstall, “The University of Washington Illustrating Compiler”, *Proc. of The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, New York, 1990, pp. 223-233.

[14] B. A. Myers, “Taxonomies of Visual Programming and Program Visualization”, *Journal of Visual Languages and Computing*, Vol. 1, 1990, pp. 97-123.

[15] B.A. Myers, R. Chandhok, A. Sareen, “Automatic Data Visualization for Novice Pascal Programmers”, *Proc. of The IEEE Workshop on Visual Languages*, IEEE Computer Society Press, New York, NY, 1988, pp. 192-198.

[16] “Pavane. Washington University”, http://swarm.cs.wustl.edu/pavane.html

[17] A. Price, R. M. Baecker, I. S. Small, “A Principled Taxonomy of software Visualization”, *Journal of Visual Languages and Computing*, Vol. 4, 1993, pp. 211-266.

[18] G. C. Roman, K. C. Cox, “Program Visualization: the Art of Mapping Programs to Pictures”, *International Conference on Software Engineering*, New York, 1992, pp. 412-420.

[19] G. C. Roman, K. C. Cox, “Abstraction in Algorithm Animation”, *Proc. of 1992 IEEE Workshop on Visual Languages*, Los Alamitos, CA, 1992, pp. 18-24.

[20] Software Visualization Group, http://www.cc.gatech.edu/ gvu/softviz/SoftViz.html

[21] J. T. Stasko, J.B. Domingue, M.H. Brown, B.A. Price (eds), *Software Visualization. Programming as a Multimedia Experience*. The MIT Press, Massachusetts, 1998.

[22] J. T. Stasko, C. Patterson, “Understanding and Characterizing Software Visualization Systems”, *Proc. of IEEE Workshop on Visual Languages*, Seattle, WA, 1992, pp. 3-10.

[23] J. T. Stasko, “Animating algorithms with XTANGO”, *SIGACT News*, Vol. 23, Iss. 2, 1992, pp. 67-71.

[24] J. T. Stasko, “The Path-Transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces”, Journal of Visual Languages and Computing, Vol. 1, No. 3, 1990, pp. 213-236.

GeoWin
A Generic Tool for Interactive Visualization of Geometric Algorithms

Matthias Bäsken and Stefan Näher[⋆]

FB IV – Informatik
Universität Trier
Trier, Germany

Abstract.

This paper introduces GeoWin, a generic visualization tool for geometric algorithms. GeoWin can be easily interfaced with todays standard geometry software libraries like LEDA and CGAL. By supporting the generic programming approach, it can be adapted to user-defined geometric objects and data types.

1 Introduction

Computational geometry is an area where the visualization and animation of programs is a very important tool for the understanding, presentation, and debugging of algorithms, and the animation of geometric algorithms is mentioned among the strategic research directions in computational geometry. It is thus not surprising that increasing attention has been devoted to algorithm visualization for computational geometry. In this paper we introduce *GeoWin*, a generic tool for the interactive visualization of geometric algorithms. *GeoWin* is a C++ data type which can be easily interfaced with algorithmic software libraries of great importance in computational geometry such as CGAL [5] and LEDA [11]. The design and implementation of GeoWin was influenced by LEDA's graph editor *GraphWin* (see [11], chapter 12). Both data types support a number of programming styles which have shown to be useful in demonstration and animation programs. Examples are the use of *result scenes* and the *event handling* approach which is discussed in section 2.2.

Most of the animations use *smooth transitions* to show the result of geometric algorithms on dynamic user-manipulated input objects, e.g., the voronoi diagram of a set of moving points or the result of a sweep algorithm that is controlled by dragging the sweep line with the mouse (see Subsection "Event Handling").

[⋆] This work is partially supported by the ESPRIT IV LTR Project 28155 (GALIA)

2 GeoWin

2.1 The Architecture of GeoWin

A GeoWin maintains one or more geometric scenes. A geometric *scene* is a collection of geometric objects of the same type. A collection is simply either a standard C++ list (STL-list) or a LEDA-list of objects. GeoWin requires that the objects provide a certain functionality, such as stream input and output, basic geometric transformations, drawing and input in a LEDA window. A precise definition of the required operations can be found in the manual pages [15]. GeoWin can be used for any collection of basic geometric objects (often called a geometry kernel) fulfilling these requirements. Currently, it is used to visualize geometric objects and algorithm from both the CGAL and LEDA libraries.

Scenes may be visualized. The visualization of a scene is controlled by a number of geometric attributes, such as color, line width, line style, etc. A scene can be subject to user interaction and it may be defined from other scenes by means of an algorithm (a C++ function). In the latter case the scene (also called *result scene*) may be recomputed whenever one of the scenes on which it depends is modified. There are three main modes for recomputation: user-driven, continuous, and event-driven. This is discussed in more detail in section 2.2 of this paper.

GeoWin offers an interactive interface (see 3.2) and a programming interface (see 3.1). The interactive interface supports the interactive manipulation of input scenes, the change of geometric attributes, the selection of scenes for visualization. The programming interface supports the definition of scenes. We illustrate the concepts above by a short example. The following four lines illustrate the programming interface. We define a GeoWin `gw` and a list `L` of objects. We next define a scene `sc` to consist of the objects in `L`. We say that the objects in `sc` are to be displayed in blue and open the scene for interactive manipulation.

```
GeoWin gw;
std::list<object> L;
geo_scene sc = gw.new_scene(L);
gw.set_color(sc,blue);
gw.edit(sc);
```

2.2 Animation with GeoWin

In computational geometry the visualization and animation of programs is a very important tool for the understanding, presentation, and debugging of algorithms. In this section we show how to use the GeoWin data type in geometric demo and animation programs by discussing three of its main approaches to interactive visualization.

Edit and Run. In the *edit & run* approach we use GeoWin in a simple loop constructing or manipulating the input scene `sc` for an algorithm by calling `gw.edit(sc)`. If this call is terminated interactively by clicking the *done* button, it returns *true*. In this case, the algorithm is applied to the objects of `sc` and the result is displayed, e.g., by changing the visual attributes of the objects. The loop will end if the edit operation is terminated by clicking the *quit* button (in this case it returns *false*). The basic structure of an *edit and run* program is given below.

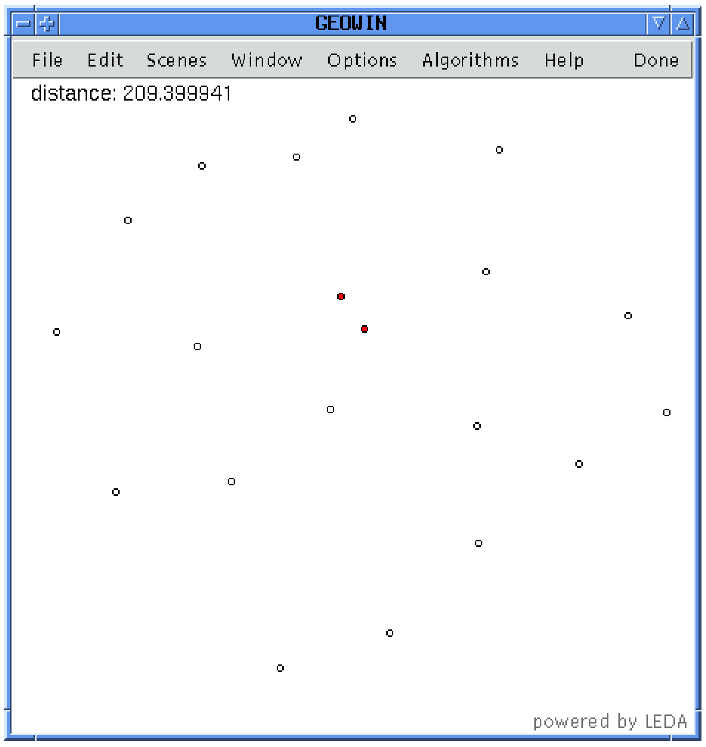
```
list<object> L;
geo_scene sc = gw.new_scene(L);

while (gw.edit(sc))
{ ALGORITHM(L);
  "display or output result";
}
```

The following code gives a complete example program for visualizing the closest pair of set of points in the plane. The program assumes that there is a `CLOSEST_PAIR` function defined that takes a list of points (of some general type `point_t`) and writes a pair of points whose Euclidian distance is minimal to the reference parameters `a` and `b`.

```
#include <LEDA/geowin.h>
#include <LEDA/geo_alg.h>

extern double CLOSEST_PAIR(const list<point_t>& L, point_t& a, point_t& b);

int main()
{
  GeoWin gw;

  list<point_t> L;
  geo_scene sc =  gw.new_scene(L);
  gw.set_color(sc,black);

  while (gw.edit(sc))
  {
    if (L.length() < 2) continue;

    point_t a,b;
    double dist = CLOSEST_PAIR(L,a,b);

    gw.set_color(sc,black);
    gw.set_obj_color(sc,a,red);
    gw.set_obj_color(sc,b,red);
    gw.message(string("distance: %f",dist));
    gw.redraw();
  }
  return 0;
}
```

The result is visualized by changing the color of the closest points to red.

This program results in an interactive visualization (Figure 1 shows a screenshot). The user may add points, delete points, pick up a point and move it around, select a subset of points and move them or rotate them with the mouse using the interactive interface of GeoWin. It is also possible to use a number of generators to generate input (we provide access to a lot of standard generators from LEDA).



distance: 209.399941

Fig. 1. Edit and Run: Screenshot of the closest pair visualization

After each user action[1] the current closest pair will be highlighted (this is done using the *set_obj_color* operations in the example program). Observe that all the dynamics comes for free to the author of the animation. It is provided by GeoWin. To switch to a demo that highlights the diameter of a point set, one only has to exchange the function `CLOSEST_PAIR`.

Result Scenes. A *result scene* is a GeoWin scene that depends on one or more *input scenes*. The dependence is defined by a function to be called for the objects of the input scenes. The objects of the result scene are just the output of this function. Whenever the input scene is modified the output scene is recomputed. In this way, it is very easy to write programs for showing the result of an algorithm on-line while the user is modifying the input of the algorithm, for example, by moving objects around, or by inserting or deleting objects of the input scenes.

[1] In the Subsections "Result Scenes" and "Event Handling" we will see how to achieve a continuous reaction of the animation.

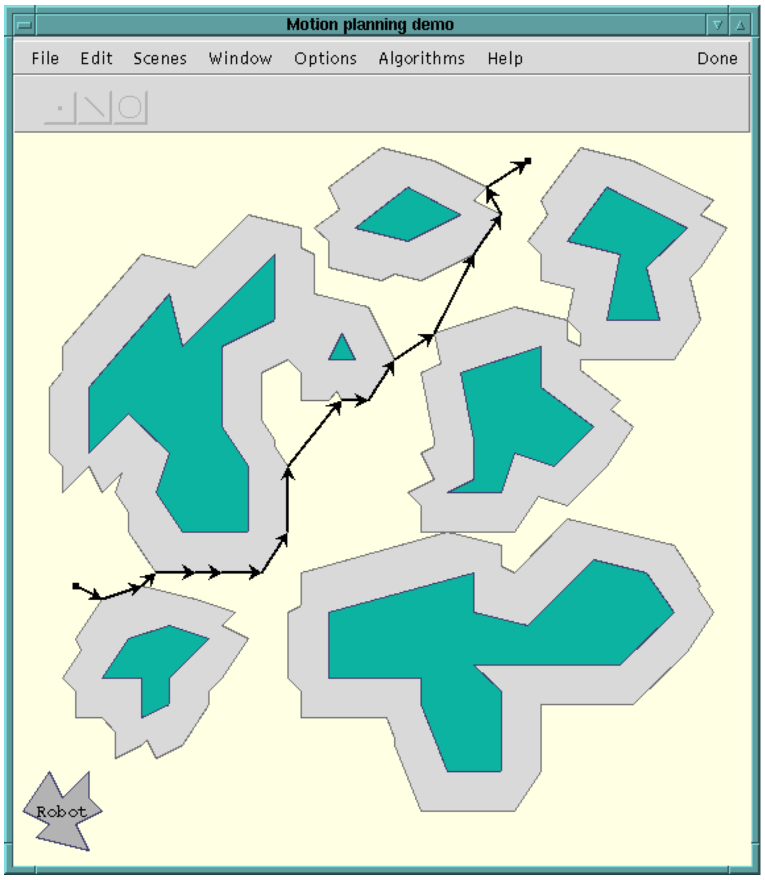


Fig. 2. Motion planning in a polygonal scene

The following piece of code shows an example program using this approach. We assume that there is a function `INTERSECT` computing the intersection points (of some type `point_t`) of a given set of straight line segments (of some type `segment_t`). Then we can create a result scene that depends on an input scene `sc_input` of points by calling `gw.new_scene(INTERSECT,sc_input)`. Many demonstration programs in LEDA and CGAL are written in this way. In particular, all algorithms working on an input set of points (e.g. all kinds of Voronoi and Delaunay diagrams) can be visualized in a single elegant program.

```
#include <LEDA/geowin.h>
#include <LEDA/geo_alg.h>

extern void INTERSECT(const list<segment_t>&, list<point_t>&);

int main()
{ GeoWin gw("Segment Intersection");
```

```
  list<segment_t> L;
  geo_scene sc_input = gw.new_scene(L);
  geo_scene sc_output = gw.new_scene(INTERSECT,sc_input);
  gw.set_color(sc_output,red );
  gw.set_visible(sc_output,true );
  gw.edit(sc_input);
  return 0;
}
```

A more advanced application of the result scene approach is shown in the screen shot of Figure 2. The underlying program shows a visualization of a motion planning algorithm in a scene of polygonal obstacles. It takes as input a scene of polygons representing a set of obstacles together with a robot polygon and a scene of two points defining the start and target position of a collision-free motion of the robot.

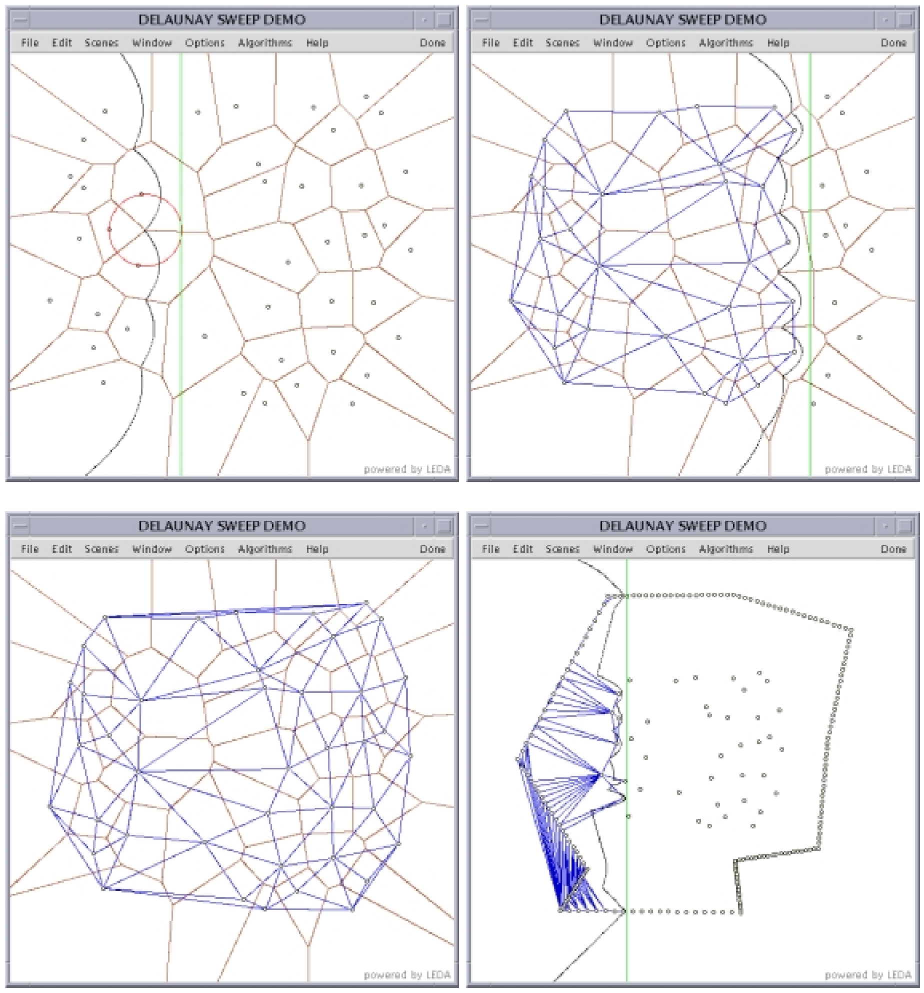
In a first step, an intermediate result scene is constructed that contains the union of the so-called Minkowski sums for each polygon with the robot. This first result scene is a general polygon (with holes) representing all points in the plane where the robot is not allowed to be positioned. Then, the second result scene (a scene of line segments) is the result of a path searching algorithm finding a collision-free motion from the start to the target point.

Event Handling. Every edit operation of the interactive interface of GeoWin has an associated *event*. For instance, creating a new object triggers a *new_object* event, deleting an object causes a *del_object* event, and moving an object around creates a *move_object* event. Application programs can handle these events by specifying corresponding call-back functions which are to be called whenever a certain event occurs. We show how to use event handling in the animation of a sweep line algorithm.

The program creates a special scene `sc_sweep` that contains a single vertical line, the sweep line, and it associates a call-back function `sweep_handler` with the `move_object` events of this scene (by calling `gw.set_move_handler(sc_sweep, sweep_handler)`). Now, during the interactive mode, the user can grab and move the sweep line with the mouse, and for each motion event the sweep handler function is called, with the relative distance vector of the motion. Note that the call-back function associated with move object events has a boolean return type. The result of this function is evaluated by GeoWin and controls whether the actual motion is really executed. In the sweep example we use this fact to prevent any backward motion of the sweep line.

```
// implementation of handlers, for instance ...
void sweep_handler(GeoWin& gw, const line& sl, double dx, double dy)
{ // move sweep line horizontally by dx do not allow backward motions
  if (dx > 0)  {
    sweep_x += dx;
    // process all events left of sweep_x ...
  }
}
```

Table 1. Animation of Fortune's Sweep Algorithm



```
int main()
{
  GeoWin gw("Sweep Demo");
  list<line>  sweep_line;
  list<point> input_points;
  sweep_line.append(line(point(0,-100), point(0,100)));

  // create input scenes ...
  geo_scene sc_sweep =  gw.new_scene(sweep_line);
  geo_scene sc_points=  gw.new_scene(input_points);
  // create result scenes for Voronoi diagram,
  // Delaunay triangulation, ...

  gw.set_move_handler(sc_sweep, sweep_handler);
  gw.edit(sc_sweep);
  return 0;
}
```

The screenshot of Table 1 shows the windows of an animation that uses the above described technique for the animation of Fortune's sweep algorithm (see [6]) for computing the Voronoi Diagram and Delaunay triangulation of a set of input points in the plane. This example allows to drag the sweep line across the plane while watching several different structures: the constructed Delaunay triangulation, the shore line of parabolic arcs, and the circle events of the sweep. These data structures are stored in different scenes of GeoWin. In the upper pictures we see the progress of the animation. We have switched off the visibility of the Delaunay triangulation scene in the upper left picture. In the lower left picture the construction of the Delaunay triangulation (that is the dual graph of the Voronoi Diagram) has finished. In the lower right picture we have zoomed into our input point set, and using a generator provided by GeoWin a number of points on a polygon were added to the input point scene. Then the sweep animation was restartet using the interactive interface, this time with switched off visibility for the Voronoi Diagram.

3 Details

3.1 The Programming Interface

In this section we will give a short overview for some groups of operations of the data type *GeoWin*. The complete list of operations is given in the user manual [15].

Group	**Description**
Main operations	used for scene creation and starting the interactive mode
Window operations	zooming, redraw, GUI initialization
Scene operations	manipulation of the various scene parameters
I/O operations	import and export of geometric data
View operations	zooming
Parameter operations	manipulation of global parameters
Event handling operations	association of user-defined event handlers with supported events

The programming interface allows us to create scenes of geometric objects, link scenes for visualization purposes and set event handling functions. It also provides the possibility to change the user interface and to add object generators and user-written I/O - functionality for input and output of geometric data in various popular formats (like VRML or GIS data formats).

3.2 The Interactive Interface

The interactive interface of a GeoWin `gw` is started by calling `gw.edit()` or `gw.edit(sc)`. The first variant opens a GeoWin with an empty set of scenes and allows the user to create and activate scenes from a menu. The second variant makes the supplied scene `sc` the active scene.

At the top of the main window there is a default menu bar containing menus for *File* and *Edit* operations, *Scenes* and *Window* setup, and other *Options*. This default menu can be changed and extended by user-defined menus and buttons.

In the same way, the default actions associated with mouse and keyboard events occurring in the drawing area of the window can be replaced or extended by user-defined actions.

In the user interface we provide the possibilty for output in postscript format for the geometric data, access to a simple 3d viewer based on the LEDA *d3_window* class ([11]) and we can set a lot of global and scene parameters (for instance graphical attributes like colors and background texture).

It also gives us access to the object generators and we have different options for object input (mouse input, textual input).

We get also access to some standard geometric algorithms that can be applied to the geometric data stored in scenes to create new scenes (an example is the convex hull algorithm computing for a set of input points stored in a scene the smallest convex polygon containing all points; calling this algorithm will create a new scene storing the output polygon).

Table 2 shows a few screenshots of the interactive interface. We see the output of the buildin convex hull algorithm and a 3d view of it. In the lower pictures one can see a scene of LEDA polygons and an options menu for a scene.

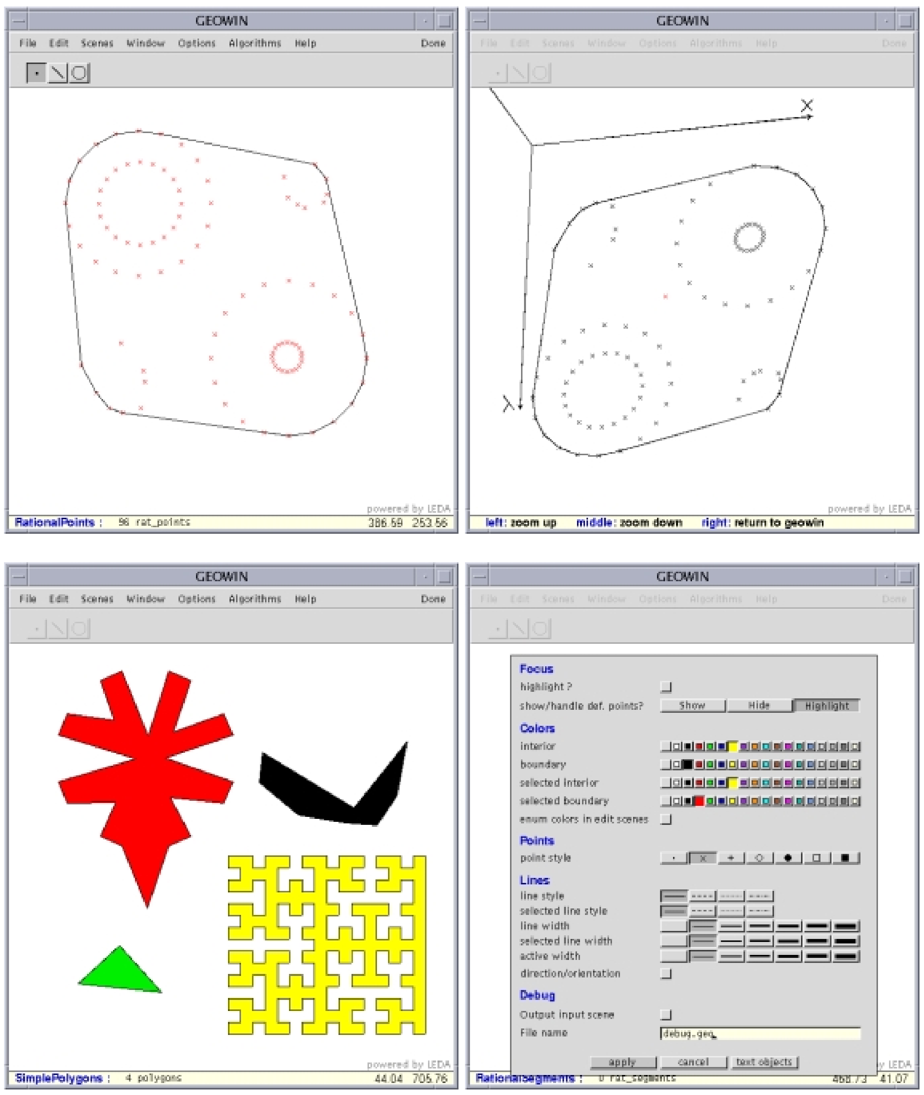
4 Genericity

GeoWin is generic and can be used with any geometry system that satisfies a small set of prerequisites. An instance `gw` of the data type *GeoWin* is an editor that maintains a collection of *scenes*. Each scene in this collection has an associated *container* of geometric *objects* whose members are displayed according to a set of visual attributes (color, line width, line style, etc.). A scene can be *visible* or un-visible and one of the scenes in the collection can be *active*. The active scene receives all editing input and thus can be manipulated through the interactive interface of GeoWin (see section 3.2).

Both the container type and the object type have to provide a certain functionality. The container type must implement the STL list interface ([12]), in particular, it has to provide STL-style iterators, and for any object type T the following functions and operators have to be defined before it can be used in GeoWin.

- I/O operators for streams, LEDA windows, and LEDA postscript files

Table 2. The interactive interface


GEOWIN
File Edit Scenes Window Options Algorithms Help Done
powered by LEDA
RationalPoints : 96 rat_points 386.59 253.56
left: zoom up middle: zoom down right: return to geowin
SimplePolygons : 4 polygons 44.04 705.76
RationalSegments : 0 rat_segments 468.73 41.07
apply
cancel
text objects

Focus
highlight ? ☐
show/handle def. points? Show Hide [✓ Highlight]
Colors
interior
boundary
selected interior
selected boundary
enum colors in edit scenes ☐
Points
point style
Lines
line style
selected line style
line width
selected line width
active width
direction/orientation ☐
Debug
Output input scene ☐
File name debug.geo

```
ostream& operator<<(ostream&, const T&);
istream& operator>>(istream&, T&);
window& operator<<(window&, const T&);
window& operator>>(window&, T&);
ps_file& operator<<(ps_file&, const T&);
```

– translate and rotate operations

```
    void Translate(T& o, double dx, double dy);
    void Rotate(T& o, double x, double y, double phi);
```

– basic geometric queries

```
    bool BoundingBox(const T& o, double& x0, double& y0,
                                 double& x1, double& y1);
    bool IntersectsBox(const T& o, double x0, double y0,
                                   double x1, double y1);
```

By using the C++ template technique for generic programming, we implemented the scene data types used by GeoWin in a way, that combinations of container and object type that fulfill these requirements for containers and objects, respectively, can be associated with GeoWin scenes in a `gw.new_scene()` operation. Currently, GeoWin is used to visualize geometric objects and algorithm from both the CGAL and LEDA libraries. In this context two typical examples for containers of a scene are

```
std::list<CGAL::POINT_2<cart> > LP;
leda_list<rat_circle>           LC;
leda_list<d3_point>             LD;
```

The first one uses an STL list from the standard C++ library and points from the Cartesian geometry kernel of CGAL, the second example uses a LEDA list of circles from LEDA's rational geometry kernel and the third example uses a LEDA list storing 3d points from the LEDA floating point geometry kernel. Especially when using a template library like CGAL that has not only simple geometric data types like points, segments, . . . but templated geometric kernel classes (like points with homogeneous integer coordinates, points with cartesian double coordinates, points with homogeneous LEDA arbitrary length integer coordinates, . . .) it is very important to provide a *generic* visualization tool that can be used with every kind of instantiation. It is of course also a very usable feature for developers writing their own geometric data types to have this flexibility. By providing the listed functionality they get the ability to use their own types with GeoWin.

5 Discussion and Comparison to Existing Work

There are many geometric visualization tools and systems available, e.g., GeomView [1] or GeoSheet [8], but these systems are not interactive and therefore their use in algorithm animation and debugging is very restricted. On the other hand, there exist many interactive programs and systems for the animation of geometric algorithms, see [4,14,3] for examples. However, these systems are not generic, i.e. they cannot be customized for different geometric objects or kernels. Many of them have even been developed on very special platforms or programming environments and therefore cannot be used in state-of the art software libraries or projects. Another system used in the visualization of geometric algorithms is GAWAIN [7,16]. It is interactive and works on different platforms, but

it is not generic and the usage with standard libraries like CGAL and LEDA is more difficult (however it has the advantage of Web-based visualization).
Current and future activities in the development of GeoWin include

- Improvement of the support for three-dimensional geometry
  We are currently working on the support of three-dimensional objects and algorithms. We have already implemented a prototype of a 3d-viewer and incorporated it into GeoWin. This 3d-viewer will be enhanced in the future.
- Improved support for incremental algorithms
  Currently, incremental algorithms are not very well supported. We plan to associate incremental data structures directly with result scenes to make their visualization much more effective and elegant.
- Algorithm animation
  We are working on a library of animation software for computational geometry.

GeoWin closes the gap caused by other systems by providing a generic platform for interactive visualization and algorithm animation in the area of computational geometry. In addition to the generic approach it supports an elegant and effective style of writing animation programs as presented in section 2.2 of this paper. GeoWin has shown to be very useful in many animation and visualization programs both in the LEDA and CGAL software libraries and it is used in debugging and implementing geometric algorithms in these projects.

References

1. Nina Amenta, Stuart Levy, Tamara Munzner, and Mark Phillips. Geomview: A System for Geometric Visualization. In *Symposium on Computational Geometry*, 1995.
2. Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. IEEE Workshop on Visual Languages, 1991.
3. P. de Rezende and W. Jacometti. Animation of Geometric Algorithms Using GeoLab. In *Symposium on Computational Geometry*, 1993.
4. P. Epstein, A. Knight J. Kavanagh, T. Nguyen J. May, and J.-R. Sack. A Workbench for Computational Geometry. *Algorithmica*, (11), 1994.
5. A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel: A Basis for Geometric Computation. In *Applied Computational Geometry: Towards Geometric Engineering Proceedings (WACG'96)*, pages 191–202, 1996.
6. S. Fortune. A sweep line algorithm for Voronoi diagrams. *Algorithmica*, (2):153–174, 1987.
7. Alejo Hausner and David P. Dobkin. Gawain: Visualizing Geometric Algorithms with Web-Based Animation. In *Symposium on Computational Geometry*, pages 411–412, 1999.
8. D. T. Lee, Chin-Fang Shen, and Dennis S. Sheu. Geosheet: A Distributed Visualization Tool for Geometric Algorithms. *International Journal of Computational Geometry and Applications*, pages 119–156, 1998.
9. K. Mehlhorn. *Data Structures and Efficient Algorithms.* Springer Publishing Company, 1984.

10. Kurt Mehlhorn and Stefan Näher. Leda: a library of efficient data structures and algorithms. *Communications of the ACM*, (38):96–102, 1995.
11. Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambrige University Press, 1999.
12. D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 1996.
13. J.-R. Sack and J. Urrutia. *Handbook of Computational Geometry*. North Holand, 2000.
14. Peter Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *Proc. Computational Geometry: Methods, Algorithms and Applications*, volume 553 of *Lecture Notes Comput. Sci.*, pages 187–202. Springer-Verlag, 1991.
15. Algorithmic Solutions. The LEDA User Manual.
16. A. Tal and D.P. Dobkin. Visualizations of geometric algorithms. *IEEE Transactions on Visualization and Computer Graphics*, (1), 1995.

Algorithm Animation Systems for Constrained Domains

Ayellet Tal

Department of Electrical Engineering
Technion – Israel Institute of Technology
Haifa, Israel
ayellet@ee.technion.ac.il
http://www.ee.technion.ac.il/~ayellet

Abstract.

This paper presents a conceptual model for designing an algorithm animation system for constrained domains. We define a hierarchy of users and a model for supporting each type of users. The hierarchy includes naive programmers, advance programmers, end users, and groups of end users. This paper also describes a few systems that realize the conceptual model within two domains: the domain of computational geometry and the domain of distributed algorithms.

1 Introduction

A major challenge in the area of algorithm animation is how to build systems which significantly facilitate the creation of algorithm visualizations regardless of the algorithms' complexity. One possible solution is to restrict the domain the algorithm animation system is designed for. In a constrained domain, knowledge regarding the type of objects and the type of operations prevalent in this domain, can be embedded into the system. As a result, the system can provide built in ways to visualize these objects and to animate the operations on them. Thus, large parts of the programmer's tasks can be automated.

This automation allows the programmer to be free from having to design and implement the visual aspects of the animation. Instead, the system makes decisions about how graphics is done. This is the major departure from general-purpose algorithm animation systems [3,4,5,6,18,19] where no assumptions are made regarding the building blocks that make up the animations. In this latter case, the programmer needs to specify the exact shape of each object and the exact transition each object goes through during the animation.

Freeing the programmer from implementing the visual aspects has a couple of benefits. The animation can be produced very fast, usually in a matter of days or even hours, since large parts of the process are left for the system to resolve. Moreover, this is done regardless of the complexity of the algorithm being visualized. This is an important point because complicated algorithms are those that gain the most from visualization.

As expected, the major disadvantage of automation is a limited flexibility. Not every algorithm can be animated and not every animation can be generated for a given algorithm. Often, users want to have a say in how the animation should look. Choices made by the system might not necessarily be appealing to the user, who might find an automatic system too restrictive. We will show in the sequel that a certain amount of flexibly can be obtained nevertheless.

In our conceptual model we define a hierarchy of users. Naive programmers care only about the contents of the visualization and are not concerned with the presentation aspects. Advanced programmers want, in addition, to be able to easily modify and extend various visualization aspects. End users experiment with an algorithm to understand its functioning, and should be able to run the animation as an interactive experience. Finally, groups of end users should be able to collaborate with each other.

This conceptual model has been realized in three algorithm animation systems built for two constrained domains. GASP [21] and GASP-II [17] were built for the domain of computational geometry; VADE [12] was built for the domain of distributed algorithms. These two domains were chosen because they represent domains in which algorithms are difficult to comprehend.

In the geometric domain even the simple task of imagining in one's mind a three-dimensional geometric construction can be hard. In many cases the dynamics of the algorithm must be understood to grasp the algorithm and even a simple animation can assist the geometer. Degeneracies and robustness problems, which are common in geometry, add to the complexity of programming and debugging geometric code. The visual nature of geometry makes it one of the areas of computer science that can benefit greatly from visualization.

Distributed algorithms are difficult to understand due to the added complexity of the inter-process communication and synchronization. Many activities occur concurrently at various sites. Moreover, the activities depend on each other in many ways. Each state depends not only on the individual process, but also on the messages arriving from other processes. Visualization can give some insight to the way this inter-process communication takes place.

We will show below that though the systems differ, they follow the same conceptual model. In the next section we describe the general conceptual model. In Section 3 we present the realization of the model in the domain of computational geometry. In Section 4 we discuss the realization of the model in the domain of distributed algorithms. We conclude in Section 5.

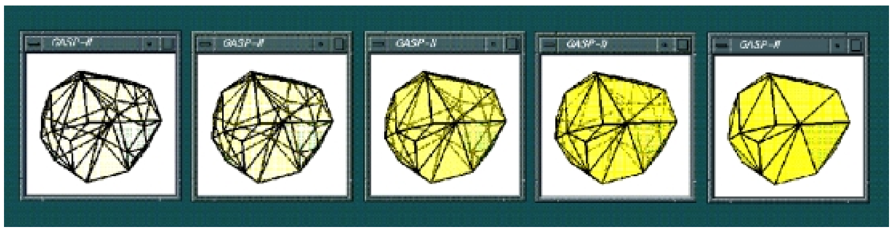
2 Conceptual Model

We define four sets of users of any algorithm animation system. They are naive programmers, advanced programmers, end users, and groups of end users. Naive programmers can choose to be isolated from any decisions of a graphical nature and concentrate solely on the contents of the animation. Advanced programmers want to be able to easily modify and extend various visual aspects of the animation in order to produce animations which better fit what the programmers think is most useful. End users experiment with an algorithm animation in order to understand the algorithm's functioning. They should be able to run the animation as an interactive experience. Finally, groups of end users wish to collaborate.

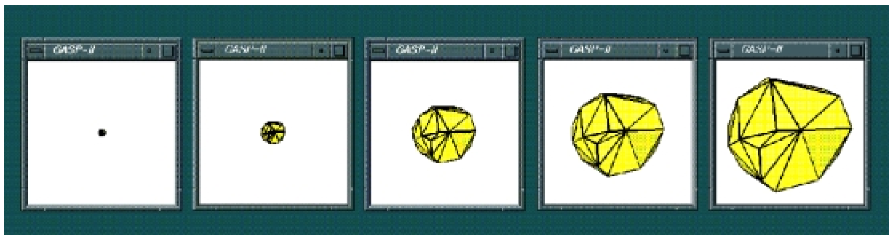
Accordingly, the algorithm animation system should provide an appropriate interface for each of these users. Domain-dependent libraries should be provided for naive programmer. An external graphical user interface should be given to the advanced programmers in order to facilitate the modification of the visual aspects. An interactive environment should let end users run the animation. Finally, tools for collaboration should be provided for groups.

In order to allow naive programmers to generate animations quickly, we distinguish between what is being animated and how it is animated. The naive programmer need only specify what is being animated and need not be concerned with how to make it happen on the screen. For example, the creation of an object is to be distinguished from the way it is made to appear through the animation. It can be created by fading into the scene, by traveling into its location, by scaling up from a point to its full size, etc.

The libraries provided by the algorithm animation system contain a set of building blocks relevant to the domain. The naive programmer should only write short snippets of code that contain calls to functions of the system's libraries. This code includes the contents of the animation and defines its structure. It does not contain the visual aspects of the animation. The algorithm animation system generates an appropriate animation from this code. For instance, only three lines of code are necessary to produce the animation of the creation of the polyhedron shown in Figure 1(a) (or the animation shown in Figure 1(b)).



(a)



(b)

Fig. 1. The creation of a polyhedron (two possible styles)

To enable that, the system's libraries support the objects prevalent in the domain and the common operations on them. In fact, several visualizations, or *styles*, are supported for each operation, one of which is the default chosen by the animation system. If the advanced programmer wishes to change any of the visual aspects of the animation, this programmer can do it through a graphical interface, called the *style panel*. There is no need to modify or write any code.

Note that the animation is still generated automatically by the animation system. But, a different animation is produced, to better reflect the programmer's taste or to better illustrate a specific algorithm. The ease of generating various animations allows advanced programmers to easily experiment with many visualizations for the same algorithm.

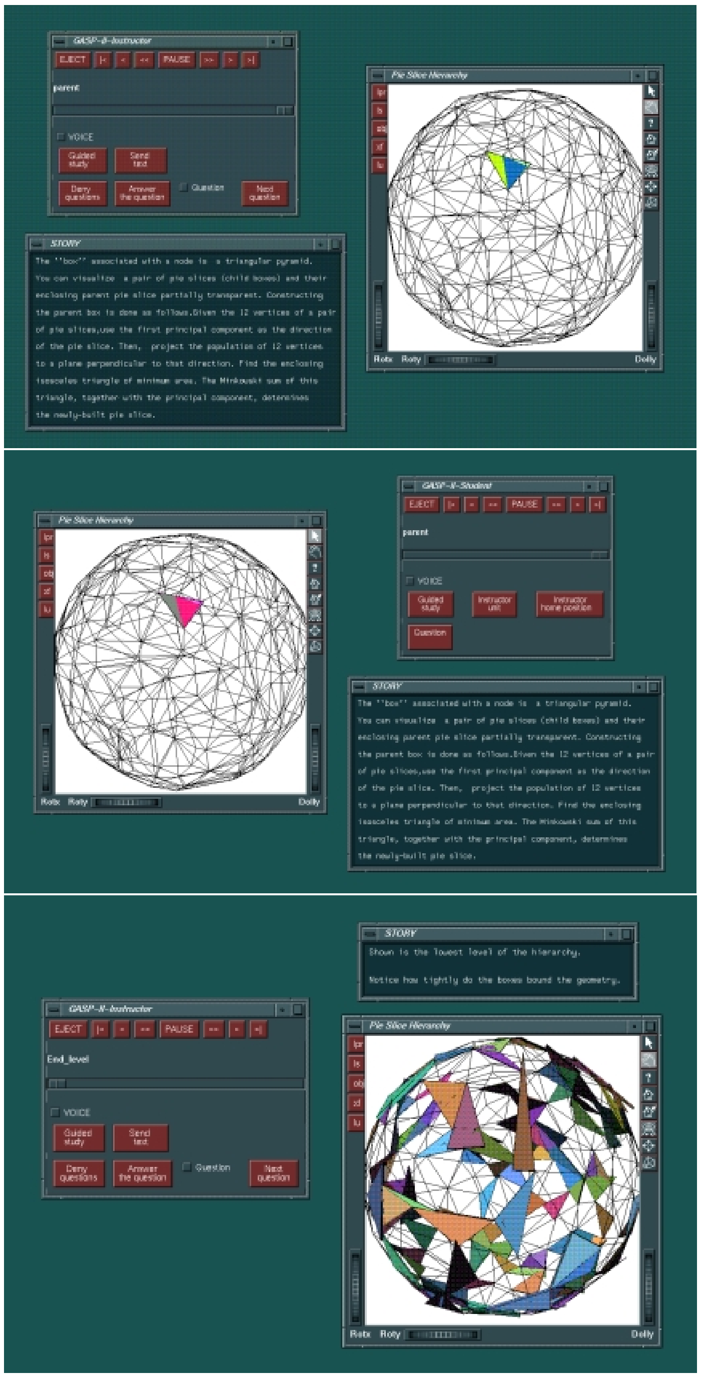
Figure 1 illustrates two of the possible styles for creating a polyhedron. In Figure 1(a) the polyhedron is created by fading into the screen, while in Figure 1(b) it is created by scaling up from a point to its full size. Changing one field in the style panel allows the creation of 1(b) instead of 1(a).

End users can experiment with an algorithm animation in an environment such as the one illustrated in Figure 2(a). The environment consists of a control panel through which the execution of the animation is controlled, animation windows where the algorithm runs, and a text window which contains explanations which accompany the animation. It is possible to run the animation forwards and backwards, to fast-forward through some parts of the animation and single-step through others.

Collaboration is useful both in the work place and in electronic classrooms where distance learning is gaining more popularity. In this environment the participants sit each in front of his or her computer and they collaboratively explore an algorithm while viewing its dynamic behavior.

In our model, two modes of distributed visualization are supported: independent visualization and collaborative visualization. In the first mode, each participant controls the animation running on his or her machine and no synchronization is applied. In collaborative visualization, the animations running on all the machines are synchronized with the animation of the instructor. In this case, the participants are more limited. They cannot fast-forward or rewind the animation whenever they wish to. However, they can still modify the style of the animation they view. This is illustrated in Figure 2 where three snapshots of screens taken at the same time are shown. The three participants, in this case an instructor and a couple of students, work collaboratively, yet each is viewing a different animation style. In particular, different colors and transparency values are chosen by the users. Moreover, the user at the bottom views a different unit of the algorithm, since this user rewound the algorithm to a previous atomic unit.

Any participant in the group can initiate an electronic discussion at any point during a collaborative session. An electronic discussion is more than a text exchange since the text is accompanied with an appropriate visualization. This is the way students can ask questions in electronic classrooms and programmers who work together can share ideas.



The ''box'' associated with a node is a triangular pyramid. You can visualize a pair of pie slices (child boxes) and their enclosing parent pie slice partially transparent. Constructing the parent box is done as follows.Given the 12 vertices of a pair of pie slices,use the first principal component as the direction of the pie slice. Then, project the population of 12 vertices to a plane perpendicular to that direction. Find the enclosing isosceles triangle of minimum area. The Minkowski sum of this triangle, together with the principal component, determines the newly-built pie slice.

The ''box'' associated with a node is a triangular pyramid. You can visualize a pair of pie slices (child boxes) and their enclosing parent pie slice partially transparent. Constructing the parent box is done as follows.Given the 12 vertices of a pair of pie slices,use the first principal component as the direction of the pie slice. Then, project the population of 12 vertices to a plane perpendicular to that direction. Find the enclosing isosceles triangle of minimum area. The Minkowski sum of this triangle, together with the principal component, determines the newly-built pie slice.

Shown is the lowest level of the hierarchy.

Notice how tightly do the boxes bound the geometry.

Fig. 2. An algorithm visualization in a distributed session

3 Realization of the Model in Computational Geometry

The last few years have seen a growing awareness to the need and to the importance of visualization in computational geometry [1,7,14,17,21]. In this domain, the scenes of interest are built out of geometric objects (e.g., polygons, polyhedra, spheres, cylinders, lines and points) and displays of data structures (such as lists and trees of various forms). A standard animation in the domain is built out of operations on these objects.

GASP [21] and GASP-II [17] realize the general model described above. GASP provides the interfaces for the naive programmer, the advanced programmer and the end user. GASP-II's main contributions are the support for groups collaborating over the network and the improvement of the interface provided to advanced programmers.

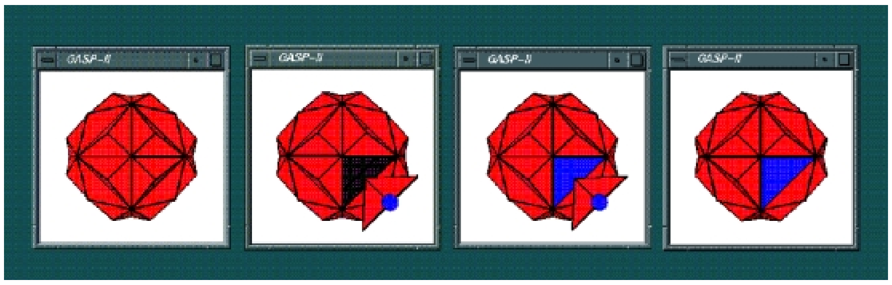
To realize the conceptual model for geometric computation, it is only required to supply appropriate libraries to be used by naive programmers. These libraries provide a rich set of animation tools for visualizing operations common in the domain. The objects contained in these libraries include three–dimensional geometric objects, two–dimensional geometric objects, combinatorial objects, views, text and titles. For example, for modifying polyhedra, the library provides functions for adding faces and vertices, removing faces and vertices, joining polyhedra, splitting polyhedra etc. The animation system can produce various visualizations for every operation, as discussed above.

Rather than going into detail describing the libraries, we will illustrate its use via a case study, which is a visualization of an algorithm for polyhedron realization for shape transformation [15]. This visualization [16] was selected as an example of an algorithm which is not basic. It is one of several animations created by GASP or GASP-II and presented at the video session of the annual symposium of computational geometry.

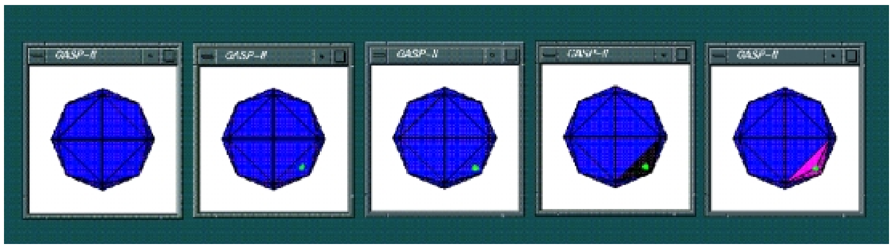
The realization algorithm proceeds in two steps, illustrated in Figures 3(a)-(b). First, each given polyhedron is iteratively simplified by removing a low–degree vertex from the $1-$skeleton graph of the polyhedron, and re-triangulating the resulting hole, until a 4–clique graph (representing a tetrahedron) results. During the visualization, which is illustrated in Figure 3(a), the selected vertex blinks in blue. Its cone of faces smoothly drives away from the polyhedron, creating a black hole in it. Next, the hole created is re-triangulated, where the new faces fade-in in blue, to distinguish them from the old (red) faces. Finally, the highlighted vertex and its cone of faces fade out.

During the second step, the process is "reversed" and the vertices are re-attached, this time in a convex fashion. During the visualization, which is illustrated in Figure 3(b), a new vertex, shown in green, appears in a base position. It smoothly drives away from the base to its final position. Then, the blue faces which were previously created while the vertex was detached, fade out. Finally, new faces attaching the vertex to the polyhedron are added, such that convexity is maintained. The new faces are colored magenta to distinguish them from the blue / red faces.

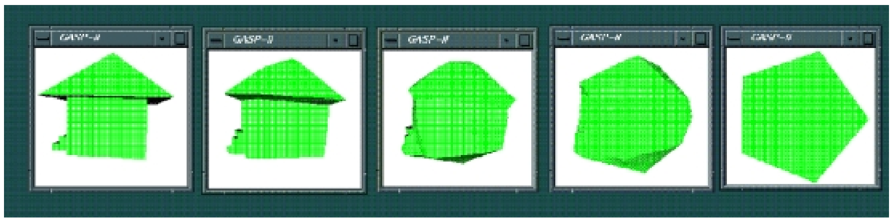
Finally, Figure 3(c) demonstrates a resulting shape transformation, where a house is transformed into an icosahedron. Each transformation is carried out by replacing the old polyhedra with a new one, until the final polyhedra is attained.



(a) The simplification phase



(b) The re-attachment phase



(c) The resulting shape transformation

Fig. 3. Animation of the realization algorithm

The major issue here is how fast it is to generate the animation. The code that generates the animation of the first step (Figure 3(a)) consists of ten lines of C code. It is shown in Figure 4. The code that generates the animation of the second step (Figure 3(b)) is twelve lines of C code. The code that generates the animation of the metamorphosis (Figure 3(c)) is four lines of C code.

The code in Figure 4 consists of calls to functions included in GASP–II's libraries. Note that no parameters of a graphical nature are required. The programmer need only specify the parameters which are related to the geometry (e.g., the vertices and the faces of the polyhedron). All the visual aspects of the visualization are left as empty slots that the algorithm animation system fills up. For instance, the code does not include the colors used, the number of frames

```
/* Remove the selected vertex */
GASP_Begin_atomic(split_atomic_name);
  GASP_Split_mesh(new_mesh_names, previous_mesh_name,
                  remove_vert_no, remove_vertices);
  GASP_Write_to_text_view("TEXT",
   "During the simplification phase, a low-degree vertex
    is removed and its vertex-graph is re-triangulated."
GASP_End_atomic();

/* Re-triangulate the hole */
GASP_Begin_atomic(add_atomic_name);
  GASP_Add_faces(new_mesh_names[0], face_no, faces);
GASP_End_atomic();

/* Remove the detached cone */
GASP_Begin_atomic(remove_atomic_name);
  GASP_Remove_object(new_mesh_names[1]);
GASP_End_atomic();
```

Fig. 4. The code of the simplification visualization

used, the fact that objects are created be fading in, etc. These parameters can later be modified in the external style panel.

4 Realization of the Model in the Domain of Distributed Algorithms

The domain of distributed algorithms is another domain where visualization in one's mind of an execution of an algorithm is highly non-trivial. Moreover, in this domain users can greatly benefit from an algorithm animation.

There is, however, an inherent difficulty in implementing an algorithm animation system within a distributed asynchronous environment. In asynchronous distributed systems it is not possible to determine the precise structure of the execution based on the views of the execution that are obtained by the processes [11]. Each process in the system can only "remember" its own actions. It can also gain knowledge about actions performed by other processes through inter-process communication. It cannot, however, compute the relative timing of the actions performed by different processes. Moreover, no form of interaction among the processes in an asynchronous system can provide precise timing information.

An algorithm animation system cannot overcome these inherent limitations. The animation processes are just more processes in the distributed environment. The goal of an algorithm animation system for distributed environments is thus to produce a visualization that reflects as closely as possible the real execution of the algorithm.

The algorithm being visualized changes dynamically and the animation system needs to receive updates of the state from different sites. Since a true instantaneous snapshot is impossible, a "possible" (or *consistent*) snapshot should be constructed. Roughly speaking, we call a snapshot of the system consistent if it describes a state that appears in a possible execution of the system that starts in the state at which the invocation of the snapshot actually happened, and ends in the state at which the snapshot algorithm completed its computation.

There exist several algorithm visualization systems for parallel or distributed algorithms [2,8,9,10,13,20]. Here we briefly present VADE [12], our visualization system which realizes the general conceptual model described above. Unlike the geometric domain, where it is sufficient to supply domain-dependent libraries in order to apply to the conceptual model, here we need also guarantee consistency.

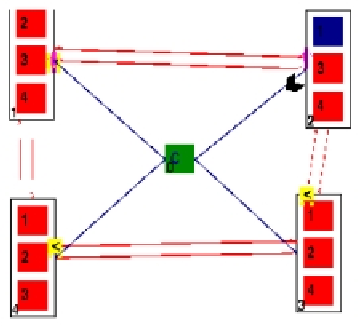
The objects common in the domain of distributed algorithms and thus the ones supported by VADE's libraries are network related objects. They include graphs, nodes, links (i.e., communication channels) and various types of messages and tokens which the nodes exchange over the links. VADE can produce a visualization for numerous operations defined on these objects. Again, the system chooses default styles for the visualization of objects and operations. The style can later be modified in an external style panel.

In [12] we describe the model of computation that our system is defined for. We also define the notion of *visualization consistency* that captures the sense in which the animation system is required to faithfully depict the execution. Using visualization consistency we can rearrange the sequence of events in a run. We discuss a couple of ways to implement visualization consistency.

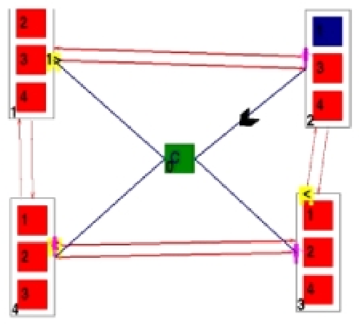
We illustrate the task of creating a visualization by describing a case study in which we simulate an Automatic Transport System (ATS). The ATS consists of a close dual-lane railroad system. The terminals are located along the railroad. Several cars can move on the railroad. In addition, there is a central control station (CCS) which has data links to all the terminals. Each terminal and each car are equipped with a display showing the stops along the cars' route. In front of each car there is an arrow-display which indicates whether it moves clockwise or counter-clockwise.

Figure 5 presents a few snapshots from the animation of this case study. Figure 5(a) shows the system immediately after a passenger arrived at Terminal 2 and pressed the Terminal 1 button. This button changes its color to blue. The terminal sends a request for a car to the CCS, animated by sending a marker over the appropriate link. Figure 5(b) illustrates the marker moving towards the CCS. In Figure 5(c) the request is received by the CCS and cars are sent from Terminal 3 and from terminal 4 to terminal 2. In Figure 5(d) the cars move towards the target terminal. Figure 5(e) illustrates the arrival of another passenger, this time at Terminal 3, which has no available cars. The appropriate button is pressed, and the request for a car is sent to the CCS. Finally, Figure 5(f) shows the arrival of a car at Terminal 2.

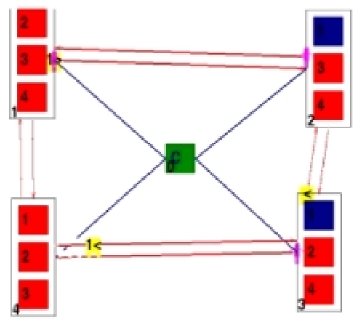
The programmer has very little work to do in order to create the above animation. This is both because the programmer need not be concerned with maintaining the consistency of the animation, and because the programmer is free from dealing with the graphical aspects of the animation which are determined by the system (and can be modified via the style panel). For instance, the code (in Java) animating the pressing of a button is shown in Figure 6.



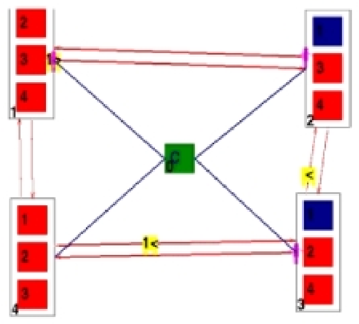
(a) Terminal 2 requests a car



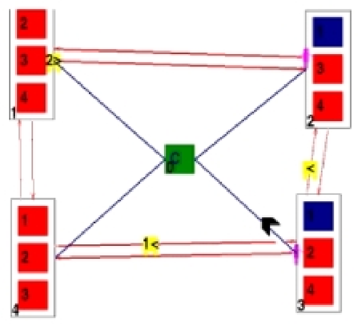
(b) A marker is sent to the CCS



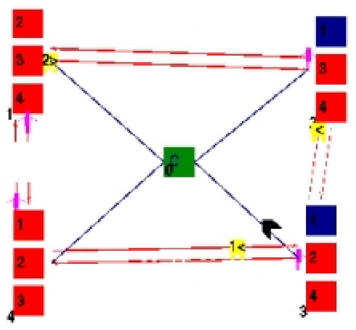
q
(c) The CCS received the request and cars are sent



(d) The cars move

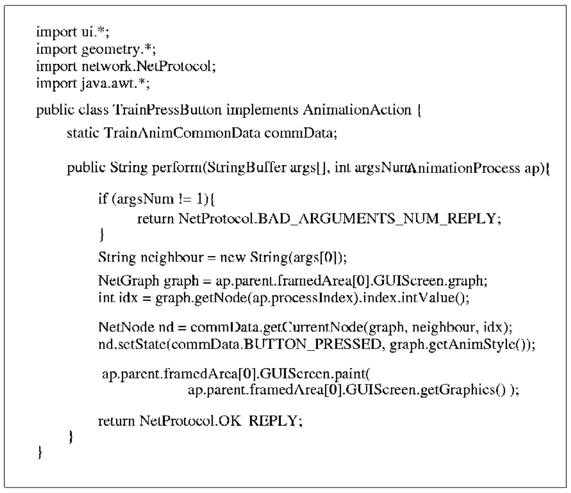


(e) Terminal 3 is asking for a car



(f) The car from Terminal 3 is arriving

Fig. 5. Snapshots from the ATS animation



```
import ui.*;
import geometry.*;
import network.NetProtocol;
import java.awt.*;

public class TrainPressButton implements AnimationAction {
    static TrainAnimCommonData commData;

    public String perform(StringBuffer args[], int argsNumAnimationProcess ap){

        if (argsNum != 1){
            return NetProtocol.BAD_ARGUMENTS_NUM_REPLY;
        }
        String neighbour = new String(args[0]);
        NetGraph graph = ap.parent.framedArea[0].GUIScreen.graph;
        int idx = graph.getNode(ap.processIndex).index.intValue();

        NetNode nd = commData.getCurrentNode(graph, neighbour, idx);
        nd.setState(commData.BUTTON_PRESSED, graph.getAnimStyle());

         ap.parent.framedArea[0].GUIScreen.paint(
                    ap.parent.framedArea[0].GUIScreen.getGraphics() );

        return NetProtocol.OK_REPLY;
    }
}
```

Fig. 6. The code of TrainPressButton class

5 Conclusions

We have presented in this paper a conceptual model for constructing an algorithm animation system for constrained domains. In a restricted domain, knowledge regarding the type of objects and the type of operations prevalent in that domain can be embedded into the system, thus enabling the creation of systems that let others use them comfortably.

We have also reviewed systems that realize this model in two specific domains – that of computational geometry and that of distributed algorithms. These domains have been chosen as representatives to domains which can greatly benefit from visualization. In geometry this is because of the difficulties in visualizing in one's mind three-dimensional scenes, along with extensive use of data structures which are heavily pointer-based and problems of degeneracies and robustness. Distributed algorithms are difficult to understand due to the added complexity of the interprocess communication and synchronization.

The major advantage of dealing with constrained domains is the automation it enables. This automation leads to time saving and facilitates the creation of animations for highly complex algorithms. The major disadvantage is that such algorithm animation systems are limited in scope.

In the future we intend to realize this model in other constrained domains, candidates being topology, databases and networks.

References

1. N. Amenta, S. Levy, T. Munzner, and M. Phillips, Geomview: A system for geometric visualization. *Proc. 11th Ann. ACM Symp. on Computational Geometry*, C12–C13, 1995.

2. T. Bemmerl and P. Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environment. *Journal of Parallel and Distributed Computing,* 18:118–128, 1993.
3. M.H. Brown. *Algorithm Animation.* MIT Press, 1988.
4. M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. *Computer Graphics*, 18(3):177–186, May 1992.
5. M.H. Brown, M.A. Najork. Collaborative Active Textbooks: A Web-Based Algorithm. Animation System for An Electronic Classroom. *Proceedings of the IEEE Symposium on Visual Languages*, 266–275, 1996.
6. M.H. Brown, M.A. Najork, and R. Raisamo. A Java-Based Implementation of Collaborative Active Textbooks. *Proceedings of the IEEE Symposium on Visual Languages*, 23–26, 1997.
7. A. Hausner and D.P. Dobkin. GAWAIN: Visualizing geometric algorithms with Web–based animation. *, The Fourteenth Annual Symposium on Computational Geometry*, pages 411–412, 1998.
8. E. Kraemer and J.T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing,* 18:105–117, 1993.
9. E. Kraemer and J.T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations *Proceedings of the 8th International Parallel Processing Symposium*, 902–908, 1994.
10. E. Kraemer and J.T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1), 36–46, 1998.
11. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558-565, 1978.
12. Y. Moses, Z. Polunsky, A. Tal and L. Ulitsky. *Algorithm Visualization for Distributed Environments*, IEEE Symposium on Information Visualization '98, October 1998.
13. G.-C. Roman, K.C. Cox, D. Wilcox and J.Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *J. Visual Languages Comput.*, 3(2): 161–193, 1992.
14. P. Schorn. The XYZ GeoBench: a programming environment for geometric algorithms, *Lecture Notes in Computer Science 553* (Springer-Verlag, Berlin, 1991), pp. 187–202
15. A. Shapiro and A. Tal. Polyhedron Realization for Shape Transformation. *The Visual Computer*, 14 (8-9): 429-444, 1998.
16. M. Shneerson, A. Shapiro and A. Tal. Polyhedron Realization and its Application to Metamorphosis. *Fifteenth Annual ACM Symposium on Computational Geometry*, June 1999.
17. M. Shneerson and A. Tal. *Interactive Collaborative Visualization Environment for Geometric Computing*, Journal of Visual Languages and Computing, Vol 6, Num 6, December 2000, 615-637
18. J. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interface. *Journal of Visual Languages and Computing*, pages 213–236, 1990.
19. J. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, September 1990.
20. J. Stasko and E. Kraemer A methodology for building application-specific visualizations of parallel programs. *J. Parallel Dist. Comut.* 18)(1):258–264, 1993.
21. A. Tal and D.P. Dobkin. *Visualization of Geometric Algorithms*, IEEE Transactions on Visualization and Computer Graphics, 1(2): 194-204, 1995.

Algorithm Animation for Teaching

Rudolf Fleischer*[1] and Luděk Kučera[2]

[1] HKUST, CS Department,
Hong Kong.
rudolf@cs.ust.hk.

[2] Charles University,
Department of Applied Mathematics,
Prague, Czech Republic.
ludek@kam-enterprise.ms.mff.cuni.cz.

Abstract.

We give an overview of rules and techniques to create a good algorithm animation, with emphasis on animations that would be used when teaching algorithms. In this context, we propose that animations should in particular emphasize the visualization of correctness invariants and the complexity of the algorithms. This implies that writing a good animation must be more than just showing the graphically enhanced runtime debugging provided by most common animation systems; instead, each animation must be individually designed and programmed.

1 Introduction

We humans are well equipped with various different senses. The skin has tactile receptors, nose and tongue can smell and taste, and the ears can perceive sound. But most powerful is our visual sense. Within a fraction of a second we can extract a big amount of information from a picture: colors, shapes, their size and relative position, etc. This great visual power eventually lead to the invention of writing, i.e., certain geometric shapes were interpreted as *letters* (or *words*, as in Chinese for example). Now any possible information could be coded and transmitted by concatenating a sufficiently large number of these letters (a *text*). But whereas this kind of information coding is quite universal, decoding is often slow. Most people grasp a message much faster when seeing a picture instead of reading (or hearing) an equivalent textual description, just remember the phrase "A picture is worth a thousand words" (see Miller [25] for nice examples).

Therefore, we are in our daily life bombarded with all kinds of pictorial messages (just open an arbitrary journal at the next book shop, or compare the time you spend watching TV with the time reading a book). Education is no exception to this trend. In computer science education, the art of explaining an algorithm by (usually animated) pictures is called *program visualization* or

* This research was partially supported by Hong Kong RGC grant HKUST 6010/01E

algorithm animation. An early example of a static visual representation of a program are flowcharts, introduced by Goldstein and von Neumann [18]. The first dynamic representation was probably Knowlton's animation of dynamically changing data structures in Bell Lab's low-level list processing language [22]. In 1981, the videotape "Sorting out Sorting" by Baecker [3] represented the first purely educational algorithm animation. The video explains nine different sorting algorithms and compares their running times. For a more comprehensive survey on the history of program visualization see the book by Stasko *et al.* [36], for example Chapter 2 by Baecker and Price [2].

Since creating good animations can be a difficult and time-consuming task, researchers started thinking about ways to automate the process of animating a program, or at least to make it easier. The first such system, `Balsa` by Brown and Sedgewick [7], was soon followed by others like `Balsa-II` [4], `Zeus` [5], `Tango` [35], `UWPI` [19], `Pavane` [33], `Catai` [9], `Jeliot` [24], `Vega` [20], `Leonardo` [10], or `Wave` [12], just to mention some of the more important ones. We will not describe these systems here in detail, that has been done in an excellent way in other surveys by Price et al. [30] and Demetrescu et al. [11]. Instead, we will in the next section argue that this approach of trying to generate program animations automatically is futile. Good animations must be designed and implemented by hand. We will focus on criteria most important for creating a good animation used for teaching. As Brown and Hershberger [6] said: "Creating effective visualizations is an art, not a science". In Section 3 we will give examples of carefully designed animations for teaching, like Dijkstra's shortest path algorithm, the bitonic sorter, and the Fast Fourier Transform algorithm.

2 What Is a Good Animation?

Basically, program animation systems fall into two categories (see Roman and Cox [31]; of course, there are also finer taxonomies for animation tools [27,30, 32]). In the *declarative* (or *data-driven*) approach, objects in the program are mapped onto images, and each change of the object results in an automatic update of the image. Examples of declarative animation tools are `Pavane` [33] and `Jeliot` [24]. In the *imperative* (or *event-driven*) approach, the animation is event-triggered, i.e., at certain 'important' points of the program execution extra function calls to geometric drawing rotuines must be added to the code. Examples of imperative animation tools are `Balsa` [7] and `Tango` [35]. Instead of calling graphic routines, an imperative animation could also just write graphical commands on a file. This file is then later used as input for a graphical editor. An example of such a system is `JAWAA` [29]. There are of course also tools with both imperative and declarative features, like `Leonardo` [10] for example.

Another declarative animation tool is the debugger `DDD` [38] that can produce nice pictures of the current set of variables in a program. Actually, any declarative animation tool is nothing else but a graphical debugger. And often, imperative systems are just used in a way similar to a declarative system; whenever one of the important data structures of the program changes the image

```
void ANIMInit(), ANIMInput(), ANIMDraw(), ANIMExchange();

static NAME_FUNCT fnc[] = { {"Init",     1, {VOID, (FPTR)ANIMInit}},
                            {"Input",    1, {VOID, (FPTR)ANIMInput}},
                            {"Draw",     1, {VOID, (FPTR)ANIMDraw}},
                            {"Exchange", 1, {VOID, (FPTR)ANIMExchange}},
                            {NULL,       0, NULL, NULL} };
main()
{  int n,i,j;
   int temp;
   int a[300];
   int count;

   TANGOalgoOp(fnc, "BEGIN");
   printf("Input number of elts in array\n");
   scanf("%d",&n);
   TANGOalgoOp(fnc, "Init");
   printf("Enter the elements\n");
   for (count=0; count<n; ++count)
      { scanf("%d",&a[count]);
        TANGOalgoOp(fnc, "Input", a, count, a[count]);
      }
   TANGOalgoOp(fnc, "Draw", a, n);
   for (j=n-2; j>=0; --j)
      { for (i=0; i<=j; ++i)
           { if (a[i] > a[i+1])
                { temp = a[i];
                  a[i] = a[i+1];
                  a[i+1] = temp;
                  TANGOalgoOp(fnc, "Exchange", a, i, a, i+1);
                }
           }
      }
   TANGOalgoOp(fnc, "END");
}
```

Fig. 1. The program `bubsort.c` from the Xtango distribution [37]

is updated accordingly. Figure 1 shows the Bubblesort animation `bubsort.c` that comes with the Xtango system [37]. The events that trigger the animation are just the changes of the data structure (in this case the array of numbers to be sorted). It is probably not a coincidence that a recent survey on program visualization tools focuses on their use as debugging aids [11].

Of course, the pictures of a graphical debugger describe the current state of a program much nicer than textual output. But is debugging a program, i.e., following its instructions step by step, really a good way to understand an algorithm? We usually do not ask our students to read source code when we explain an algorithm. Stepping through a program with a graphical debugger is often nice to watch (all these colorful data structures) but it is just a visualization of the source code and therefore not a potentially superior teaching aid. Indeed, many attempts to assess the usefulness of animations (created by animation tools) have produced inconclusive results (see Stasko and Lawrence [34]), although some recent studies have been more successful in establishing the usefulness of animations for teaching algorithms [21].

Petre at al. [28] noted that "supporting users in understanding an algorithm requires more than illustrating the primitive code". Indeed, do we not ask our students to think in more abstract terms or in functional units when we teach them algorithms? Fix and Sriram [15] claim that code-steppers (like `Balsa`) are not very suitable for teaching because they highlight sequential operations instead of the general structure of a program.

We would not go as far as Dijkstra who feared "permanent mental damage for most students exposed to (a certain educational program visualization software)" [13] (he did not specify the particular program visualization software he had in mind) in condemning animation tools. But, as Baecker observed, "animating programs for pedagogical purposes is not a trivial endeavor" [1]. Many authors give advice on the features of a good animation, see for example Baecker [1], Miller [25], Gloor [17] and Faltin [14]. The following rules summarize their suggestions.

– **Model:** [25]
  Any visualization needs a model that can be illustrated. Since algorithms are abstract entities without a physical reality, an appropriate model for the algorithm to be visualized should be chosen. Often, this is one of the key data structures of the algorithm (a graph exploration algorithm is usually animated by visualizing the exploration on a drawing of the graph, a tree-based dictionary is usually animated by drawing the balanced search tree, etc.).
– **Functional structure:** [14,17]
  All the functions and their interactions in an algorithm should be explained. Some people prefer a top-down approach to programming, others prefer to explain programs bottom-up. Both ways are fine, as long as the description is complete and every step of the algorithm is explained. Very important is to find the right level of abstraction. In the best of all worlds, an animation would somehow adapt to the level of user knowledge (it could gain information by interacting with the user).
  Sometimes it even might be a good idea to break the time linearity of events and show different stages of the algorithm at the same time.
– **Correctness and runtime:** [33,16]
  A good algorithm animation should show *why* the algorithm is correct and *why* it is efficient.
– **Speed:** [1]
  Different parts of an algorithms need different animation speeds. If there is a loop (like the main loop in Quicksort or Bubblesort) then it is enough to

explain it once slowly and in detail and then run it a second time (or a few times) fast to get a feeling for the dynamics of the loop.

- **Text:** [14,25]
  Whether an animation should include text explanations mainly depends on its application. A classroom animation usually does not need much text. Projected onto a screen most text is unreadable anyway, and the instructor can explain much better what is happening. If labels are unavoidable, they should be meaningful. It is much better to display values graphically (by bars of different height, for example) than to give alphanumerical values.
  On the other hand, if the animation is used in an electronic textbook for an online course then it needs lots of text explanations. If sound is available, narrative explanations (as in a classroom lecture) might be advantageous.
- **Code or Pseudocode**: (in contrast to [14])
  Algorithms should be taught on an abstract level. In particular, the students should understand them independently of any programming language. Showing code is only useful for programming courses.
  Pseudocode on the other hand is a higher-level description of an algorithm, and it can make sense to show it in parallel to the execution of a program. Except, that a good animation not necessarily just 'executes' a program, so the use of this feature might actually be limited.
- **Uniform representation:** [1]
  An algorithms course might feature many animated algorithms. A uniform outlook (same layout of information on the screen, similar arrangement of buttons, etc.) is very helpful for students. If every animation looks completely different, they need more time to adjust.
- **Simplicity**: [17,25]
  The animations should be simple, also the controls. Basic controls like 'run', 'step', 'backward' are usually sufficient. Fancy graphics should not distract from the algorithm. Color should be used carefully to code information, not to entertain. Sound can also be helpful to code information [6].
- **Data**: [1,14]
  Algorithms should be explained on small input data sets, or better on different sets of interesting input data sets. If the algorithm is understood, it can make sense to run it on larger input data sets, for example to demonstrate asymptotic running time behaviour.
- **Interactivity:** [14,17,25]
  Students presumably learn better when they can play around with an algorithm. The algorithm should be able do handle user inputs. There can be different forms of interactivity: user specified input data, user modified program code, quizzes on the behaviour of the algorithm, etc.
- **Fun:** [17]
  Students learn better if the animation keeps them interested. Avoid boring animations (like showing all 27 iterations of the main loop of Quicksort).

Using a declarative or imperative animation system, one can easily achieve goals like uniformity or interactivity. But the major goal of explaining the functional structure of an algorithm and visualizing its correctness and running time needs careful design ("even though it may be easy to animate a program, it's not so easy to produce an effective and informative animation", Brown and Hershberger [6]). The Pavane system [33] explicitly favors proof-based visualizations, and the highly successful video "Sorting out Sorting" by Baecker [3] visualizes different running times by showing a race between the nine sorting algorithms explained in the video. It is worth noting that the production of this video took three years [1] and was not just the result of the automatic animation produced by some animation system. It uses a minimalistic approach by only showing the data (and how it changes).

A good animation needs a sophisticated conceptualization [16]. The major questions in the design of a good animation [1,27] are

– What to show?
– When to update?

Mulholland and Eisenstadt [26] found empirical evidence in the development of their Transparent Prolog Machine (TPM) that novice programmers do not always benefit from seeing graphical animations because the abstract interpretation of these graphics needs some basic knowledge of algorithmics; without that experience an animation is just a nice movie but the content hidden behind the animation gets lost. Lahtinen et al. [24] go one step further and claim that students learn much more about an algorithm when *creating* an animation instead of just watching one. This is in line with Faltin's [14] SALA (Structured Active Learning of Algorithms) approach to teaching algorithms which lets students assemble complicated algorithms from simpler components in an interactive learning environment.

3 Good Teaching Animations

In this section we will focus on three rules which in our opinion are most important for a successful teaching animation — the graphical display of values, visualization of algorithm invariants, and visualization of the construction of an algorithm. They follow from our own research and implementation of algorithm animations and stress items that seem to us to be underestimated or missing in the general literature on algorithm animation.

3.1 Display Values Graphically

Alphanumerical display of values is very useful if the *exact* value is looked for (e.g., measurement of voltage, temperature, etc.). However, algorithm animations and many other areas of visualization usually need to show *relations* among different value. As examples we can mention

– number sequences, when relations of values in the sequence are important (e.g., increasing, decreasing, bitonic, randomly looking sequence);

– time evolution of a particular value, when the immediate value has to be compared to a value in one or more moments in the past or in the future (e.g., increasing or decreasing in time, oscillating).

Our experience shows that it is extremely important to display the relation of values in a graphical way in order to make the presentation easy to understand.

This rule is easy to follow and almost always followed in the case of a linear sequence of numbers that typically occurs in sorting algorithm animations. A simple horizontal row of vertical bars of length proportional to the values is sufficient.

The rule is, however, surprisingly often overlooked in only slightly more complex situations. For example, the main invariant of *binary search trees (BST)* is that values stored in nodes increase from left to right (in the usual graphical representation of a BST). Fig. 2 shows a BST presentation [23] that might significantly enhance a student's understanding of the BST search by giving an alternative graphical way of illustrating the situation. The invariant is obvious from bars located on the baseline below the corresponding nodes. It is also clear that a query, represented graphically by a horizontal line in a given height over the baseline, forces a move left from the current node represented by a vertical arrow hanging from the node.

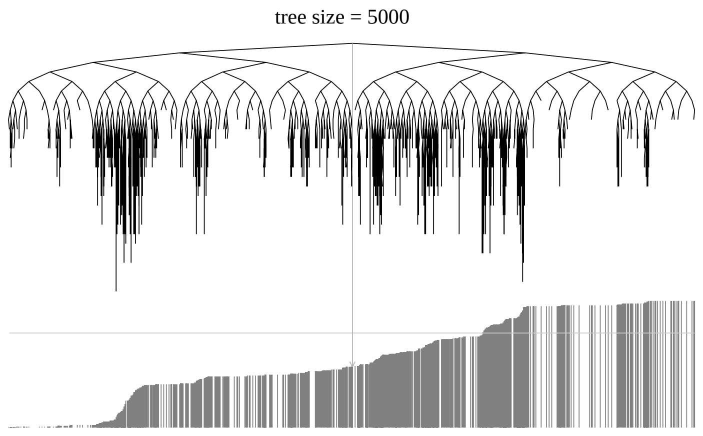


Fig. 2. A binary search tree. Only the top eight layers are shown, all deeper nodes are hidden in subtrees; the length of the bar representing a subtree indicates its depth. The lower half of the picture shows the BST invariant that node values in the tree are non-decreasing from left to right, and a horizontal bar represents the current search value.

In some cases one-dimensional structures are preferably displayed in a modified way, for example, a bitonic sequence (defined as a *cyclic shift* of a sequence that first increases and then decreases) can, due to its nature, best be recognized in a circular representation as shown in Fig. 7.

The rule is much more difficult to apply to two dimensional structures. One frequent source of such objects are graph algorithms, where the usual representation of the graph is as a planar drawing (even for non-planar graphs, where, of course, crossing edges have to be allowed). Square or rectangular matrices are another more regular source. Such matrices could also result from the time evolution of one-dimensional arrays.

No technique seems to be universally applicable in such a situation. The three-dimensional view of an object obtained from a planar structure by projecting values into the third dimension often gives very good and illustrative presentations, but in general it is difficult to determine the best angle to view the plot, and hence a user control might be necessary to "fly" over it. As a result, this solution is quite complex from the view of both a programmer and a user. Fig. 3 shows 3D views of a (node and edge) labeled planar graph used in a Dijkstra's shortest path algorithm applet [23].

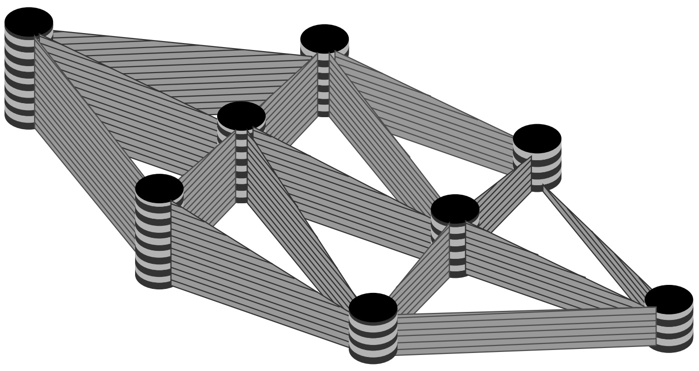


Fig. 3. Dijkstra's shortest path algorithm animation. The 3D representation of the nodes and edges of the explored part of the graph helps to better understand the algorithm.

In that example, reading the label of a node is further enhanced by composing a column from layers of different color, where the height of the column displays the value of the node label. An edge label (its length) is visualized by drawing the upper border as a decreasing line; horizontal lines are used to make it easier to see how much altitude is lost when an edge is traversed.

An additional benefit of the 3D representation in this particular application is that the difference between the unexplored part of the graph, which is merely drawn on the floorplan as a 2D object, and the part thas has been or is being processed is very clear and further helps to properly understand the method.

Alternative ways are *ad hoc* methods that represent values graphically without adding extra dimensions. It is sometimes very convenient to represent the value of an item by color, starting with black for the lowest values and then going through blue, red, and yellow to white for the highest ones, and that this method is especially advantageous if values correspond to a 'real' physical variable (pressure, voltage, or even temperature itself). The physical size of an object representing an item (matrix entry, graph node or edge, etc.) is another possibility for graphical display. For example, in a graph the diameter of a circle representing a node or the width of a line representing an edge could be proportional to the label of the object. But although such methods are much simpler to implement, they are also much less illustrative and do not give a direct impression.

3.2 Visualize Algorithm Invariants Rather than Behavior

The theory of program verification tells us that knowing invariants necessary to prove algorithm correctness is equivalent to understanding the algorithm. On the other hand, an animation of an algorithm that only shows *what* is going on without giving an idea of *why* is usually understandable just for observers that already know the algorithm's function and background ideas, and has therefore only a very limited value as a teaching tool. In order to build an animation that really teaches an algorithm it is a good idea to try to create a good visual representation of the invariants and ways that demonstrate the design ideas.

Let us return to the example of binary search trees for which many animations can be found on the Internet. Here the invariant is easy: "left/right descendants of any vertex have labels that are smaller/greater than the label of the vertex", and therefore no effort is usually paid to its *visual* presentation. An animation displaying labels of nodes as numbers drawn in the nodes enables a user to check validity of the invariant and to predict te algorithm's behavior, but it is a textual rather than a visual presentation, and therefore reading rather than seeing is necessary to grasp the invariant.

Fig. 2 shows one possible visual representation of a BST invariant [23]: nodes of BST are naturally ordered from left to right, and if label values are shown by vertical bars situated below the corresponding nodes it is clear that the sequence is increasing from left to right. Since left goes down and right goes up, if a query is visualized by a horizontal line and the current node during the search is projected vertically onto the label sequence, it is immediately obvious to which of the sons of the current node the search goes.

3.3 If Possible, Visualize an Algorithm's Construction

In certain cases it is much more desirable to explain how an algorithm has been built than how it works. This is especially a useful approach when the algorithm in question can be represented using a recursively defined linear program or combination circuit. As an illustration, two applets explaining a bitonic sorter and Fast Fourier Transform (FFT) are given [23].

A bitonic sorter is originally displayed as a black box. A recursive construction can be stepped through; Fig. 4 shows the first step which illustrates the main idea of the construction. However, classroom experience shows that it is very useful to provide the possibility to go through the whole construction sequence, see Fig. 5: not only students have no problems to draw a sorter with, say, 16 inputs, but they are able to see the recursive structure behind it and to understand its function. It is very convenient when the circuits really works at any moment of the construction, and a sorting process can be visually followed through circuit levels, i.e., stepping through a partially developped circuit (see Fig. 4) illustrates clearly how the algorithm invariant is followed.

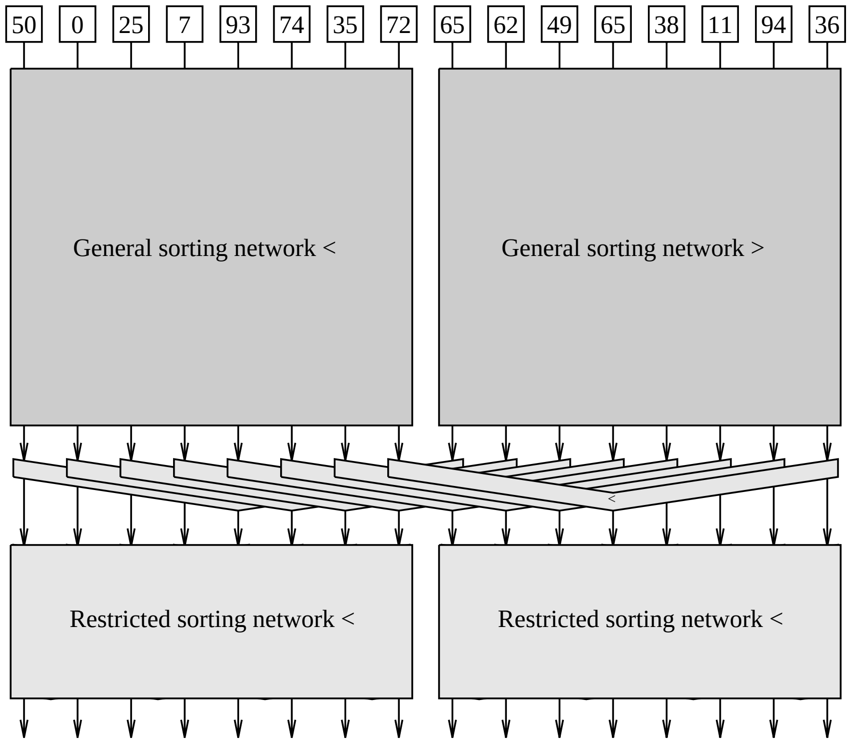

50
0
25
7
93
74
35
72
65
62
49
65
38
11
94
36
General sorting network <
General sorting network >
<
Restricted sorting network <
Restricted sorting network <

Fig. 4. Bitonic sorter (partially developped). The lower window shows the invariant that both halves of the input sequence are bitonic.

An FFT applet shows how a butterfly based FFT network is built by a step-by-step transformation from the original $N \times N$ matrix that formally defines a discrete Fourier transform. The key step of the construction consists of showing that, after a decomposition of the problem into four blocks of size $N/2 \times N/2$, there are only *two* different blocks. This can easily be visualized by moving one of the equal blocks on the screen to *graphically overlay* the other one. Fig. 6 displays a situation shortly before the complete match of the blocks. Since the second author has been making the applet available in an undergraduate algorithms course the number of students with problems understanding the FFT algorithm has decreased notably (but we admit that this might also be due to other unrelated factors).

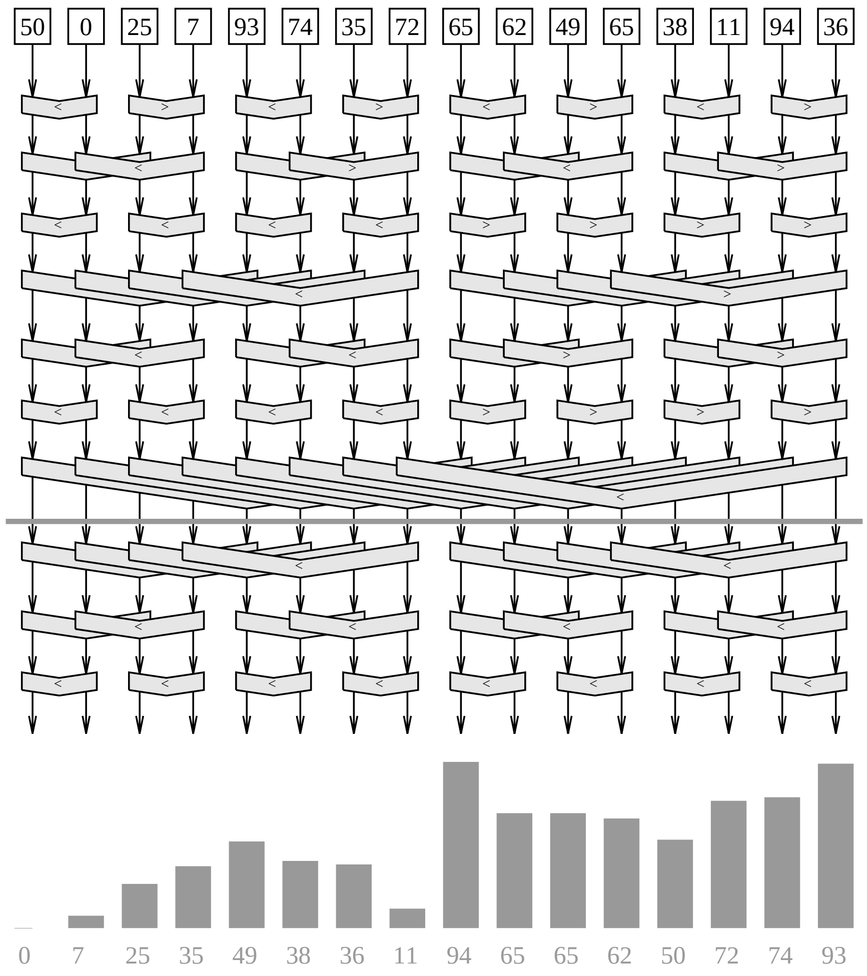


Fig. 5. Bitonic sorter (fully developped).

3.4 Could and Should a Mathematical Proof Be Visualized?

A bitonic sorting network construction uses a subnetwork of depth one that splits a bitonic sequence into two bitonic sequences, each element of the first one being smaller or equal to any element of the second one. The animation mentioned above and shown in Fig. 4 and Fig. 5 can be used to *check* that this is always true, but without giving any hint *why* this happens. A rigorous mathematical proof of the proposition is not trivial, and it is not a good idea to build a chimeric product that consists of a Java animation combined in an unseparable way with a difficult mathematical text.

It appears that one possible solution is to use what we call a 'virtual machine'. This is a visual dynamic object that is operated by a student using certain controls; his or her task is to reach the final state of a machine (corresponding

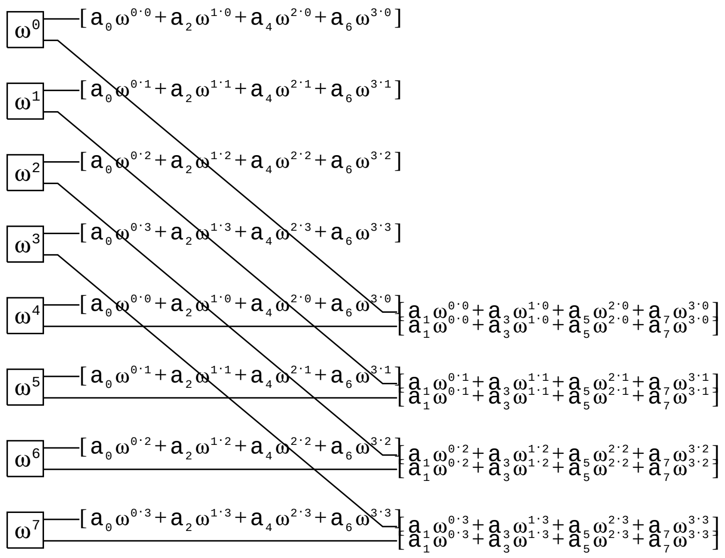

ω^0
ω^1
ω^2
ω^3
ω^4
ω^5
ω^6
ω^7
$[a_0\omega^{0\cdot0}+a_2\omega^{1\cdot0}+a_4\omega^{2\cdot0}+a_6\omega^{3\cdot0}]$
$[a_0\omega^{0\cdot1}+a_2\omega^{1\cdot1}+a_4\omega^{2\cdot1}+a_6\omega^{3\cdot1}]$
$[a_0\omega^{0\cdot2}+a_2\omega^{1\cdot2}+a_4\omega^{2\cdot2}+a_6\omega^{3\cdot2}]$
$[a_0\omega^{0\cdot3}+a_2\omega^{1\cdot3}+a_4\omega^{2\cdot3}+a_6\omega^{3\cdot3}]$
$[a_1\omega^{0\cdot0}+a_3\omega^{1\cdot0}+a_5\omega^{2\cdot0}+a_7\omega^{3\cdot0}]$
$[a_1\omega^{0\cdot1}+a_3\omega^{1\cdot1}+a_5\omega^{2\cdot1}+a_7\omega^{3\cdot1}]$
$[a_1\omega^{0\cdot2}+a_3\omega^{1\cdot2}+a_5\omega^{2\cdot2}+a_7\omega^{3\cdot2}]$
$[a_1\omega^{0\cdot3}+a_3\omega^{1\cdot3}+a_5\omega^{2\cdot3}+a_7\omega^{3\cdot3}]$

Fig. 6. Fast Fourier Transform animation. Overlaying the edge labels of two of the four blocks of output edges shows that they are actually identical.

to the input data). At the beginning a user receives hints, but later he or she should learn to work unguided. If the machine is appropriately designed a user who knows how to operate the machine also knows how to prove the original statement.

In the case of the bitonic sorter, the machine is built over a circular representation of the input bitonic sequence, see Fig.7. A user operates the machine by rotating an arrow that points to pairs of items that are compared by a separator subcircuit of a bitonic sorter. He or she is guided through a series of subtasks that must be finished and separated by pushing the 'Enter' button; the first task is to find the minimum and maximum elements of the sequence, then to observe that paths from the minimum to the maximum are monotone, and the final task is to find a proper cut of the input sequence that creates the two output bitonic sequences of the bitonic separator.

Of course, such a visual 'proof' is not a proof in a strict sense, it is only an example providing convincing evidence (although even mathematicians sometimes resort to a 'proof by convincing example' [8]). It does not deal with arbitrary bitonic sequences, even though we believe that it might be made truely general using, for example, a graphical symbol standing for a *general* increasing sequence, i.e., an abstract notion that represents a collection of particular similarly looking objects. However, if a mathematical proof is used not to demonstrate correctness of a given proposition but to explain *why* the proposition is correct, such 'proofs' (or convincing examples) based on inductions from typical cases and using

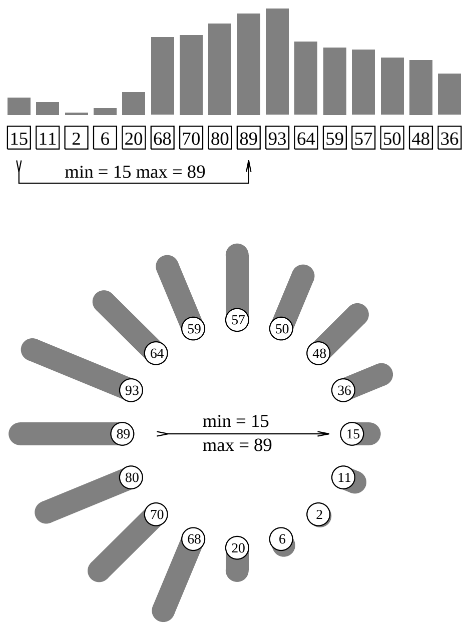


Fig. 7. Bitonic separator. The circular representation of the array makes it easier to understand the properties of a bitonic sequence.

visual virtual machines seem to be equally good and in many cases easier to understand.

4 Conclusions

We summarized advice on the features of a good animation as found in the recent literature and stressed those rules that, in our view, are the most important. We believe that it is not sufficient to create a "movie" illustrating the functioning of an algorithm, but it is also necessary to use graphical tools to explain why the algorithm works (which in most cases means the presentation of algorithm invariants, as used in a correctness proof), to visualize its complexity (time, memory, chip area, etc.), and to show how it was constructed. In some cases, a good visual presentation requires a completely different approach that is not a simple translation of a standard text-book description moved to the screen. In particular, there is no blueprint how to create a good animation. For each algorithm, we must find out where the conceptual difficulties of the algorithm are hidden and then we must decide which weighting of the different criteria of Section 2 can how be animated to show these difficulties in the best possible way.

References

1. R. Baecker. Sorting out Sorting: A case study of software visualization for teaching computer science. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 24, pages 369–381. The MIT Press, Cambridge, MA, and London, England, 1997.
2. R. Baecker and B. Price. The early history of software visualization. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 2, pages 29–34. The MIT Press, Cambridge, MA, and London, England, 1997.
3. R. M. Baecker. Sorting out sorting, 1983. Narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review No. 7, 1983.
4. M. H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, 1988.
5. M. H. Brown. Zeus: A a system for algorithm animation and multi-view editing. In *Proceedings of the 7th IEEE Workshop on Visual Languages*, pages 4–9, 1991.
6. M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25:52–63, 1992.
7. M. H. Brown and R. Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, 1984.
8. N. G. de Bruijn. Sorting by means of swapping. *Discrete Mathematics*, 9:333–339, 1974.
9. G. Cattaneo, U. Ferraro, G. F. Italiano, and V. Scarano. Cooperative algorithm and data types animation over the net. In *Proceedings of the IFIP 15th World Computer Congress on Information Processing (IFIP'98)*, pages 63–80, 1998. System home page: `http://isi.dia.unisa.it/catai`.
10. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. System home page: `http://www.dis.uniroma1.it/~demetres/Leonardo`.
11. C. Demetrescu, I. Finocchi, G. F. Italiano, and S. Näher. Visualization in algorithm engineering: Tools and techniques. In *Experimental Algorithics — The State of the Art*. Springer-Verlag, Heidelberg. To appear.
12. C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing algorithms over the Web with the publication-driven approach. In *Proceedings of the 4th Workshop of Algorithms and Engineering (WAE'00)*, 2000.
13. E.W. Dijkstra. On the cruelty of really teaching computing science, The SIGCSE Award Lecture. *Communications of the ACM*, 32(12):1398–1404, 1989.
14. N. Faltin. Aktives Lernen von Algorithmen mit interaktiven Visualisierungen. In K. Mehlhorn and G. Snelting, editors, *Neue Horizonte im neuen Jahrhundert. 30. Jahrestagung der Gesellschaft für Informatik Berlin, 19.-23. September 2000*, pages 131–137. Springer-Verlag, Heidelberg, 2000. `http://www-cg-hci-e.informatik.uni-oldenburg.de/~faltin/ALernAlg`.
15. V. Fix and P. Sriram. Empirical studies of algorithm animation for the selection sort. In W. Gray and D. Boehm-Davis, editors, *Empirical Studies of Programmers: 6th Workshop*, pages 271–282. Ablex Publishing Corporation, Norwood, NJ, 1996.
16. P. A. Gloor. Animated algorithms. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 27, pages 409–416. The MIT Press, Cambridge, MA, and London, England, 1997.

17. P. A. Gloor. User interface issues for algorithm animation. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 11, pages 145–152. The MIT Press, Cambridge, MA, and London, England, 1997.
18. H. H. Goldstein and J. von Neumann. Planning and coding problems of an electronic computing instrument. In A. H. Traub, editor, *von Neumann, J., Collected Works*, pages 80–151. McMillan, New York, 1947.
19. R. R. Henry, K. M. Whaley, and B. Forstall. The University of Washington Program Illustrator. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, pages 223–233, 1990.
20. C. A. Hipke and S. Schuierer. VEGA: A user centered approach to the distributed visualization of geometric algorithms. In *Proceedings of the 7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 110–117, 1999.
21. C. Kehoe, J. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, 2001.
22. K. Knowlton. Bell telephone laboratories low-level linked list language, 1966. 16-minute black and white film, Murray Hill, N.J.
23. L. Kucera. Homepage. `http://www.ms.mff.cuni.cz/acad/kam/kucera`.
24. S. P. Lahtinen, E. Sutinen, and J. Tarhio. Automated animation of algorithms with Eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.
25. B. P. Miller. What to draw? When to draw? An essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, 18:265–269, 1993.
26. P. Mulholland and M. Eisenstadt. Using software to teach computer programming: Past, present and future. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 26, pages 399–408. The MIT Press, Cambridge, MA, and London, England, 1997.
27. B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
28. M. Petre, A. Blackwell, and T. Green. Cognitive questions in software visualization. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 30, pages 453–480. The MIT Press, Cambridge, MA, and London, England, 1997.
29. W. C. Pierson and S. H. Rodger. Web-based animation of data structures using JAWAA. In *29th SIGCSE Technical Symposium on Computer Science Education*, 1998. System home page: `http://www.cs.duke.edu/csed/jawaa/JAWAA.html`.
30. B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
31. G. C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22:25–36, 1989.
32. G. C. Roman and K. C. Cox. A taxonomy for program visualization systems. *Computer*, 26:11–24, 1993.
33. G. C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. PAVANE: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
34. J. Stasko and A. Lawrence. Empirically assessing algorithm animations as learning aids. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 28, pages 419–438. The MIT Press, Cambridge, MA, and London, England, 1997.

35. J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.
36. J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, Cambridge, MA, and London, England, 1997.
37. Xtango. A general purpose algorithm animation system. `http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html`.
38. A. Zeller. Ddd — data display debugger, version 3.3, 2000. `http://www.gnu.org/software/ddd`.

Chapter 2
Software Engineering

Introduction

Rym Mili and Renee Steiner

Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688,
Richardson, TX 75083-0688, USA
rmili@utdallas.edu,
renee.steiner@intervoice-brite.com

Software engineers have long needed a way to understand complex software systems during all phases of the lifecycle. This need is driven by the fact that, in Software Engineering, there is ample evidence that a clear and visual representation of a software product can significantly enhance its understandability and reduce the lifecycle cost. For instance, when large volumes of data or text are to be understood or analyzed, it is often the case that a simple visual representation of the information allows the user to quickly and accurately detect discrepancies caused by confusing software documents. This early understanding reduces the person-months needed to take the project through successful system implementation.

However, in order to discuss Software Visualization (SV) even briefly, we need a common nomenclature. Several definitions of SV were given in the literature [17] [4] [27] [25] [33]. In this paper, we approach SV from a Software Engineering perspective, and hence we propose the following definition: software visualization is a representation of computer programs, associated documentation and data, that enhances, simplifies and clarifies the mental representation the software engineer has of the operation of a computer system. A mental representation corresponds to any artifact produced by a software engineer that organizes his or her concept of the operation of a computer system. Hence, the main implications of our definition are: 1) we do not restrict the term software to mean computer programs; 2) we do not impose any restrictions on the term visualization: it can be textual or graphical provided it meets the criteria established in the definition.

There is a wide range of SV tools on the market today and they differ in regards to the type of visualization they provide, and the activity they target. In order to synthesize past achievements and define a framework for future work, we present a survey as well as a classification of SV tools in software engineering. In section 1, we present a set of attributes that can be used to classify SV tools. We have divided existing tools into two broad families which are discussed in sections 2 and 3. In section 4, we present a synthetic assessment of the two families of tools, and suggest directions for future work.

1 Classifying Software Visualization Tools

SV tools are a necessary part of the software engineer's daily regime. When used in the early phases of software development, they yield lower costs and better results in the implementation and maintenance phases. When used in the later stages, they help developers maintain and enhance existing source code bases. In the next section, we begin to characterize some tools that are produced commercially as well as academically.

1.1 Characterizing SV Tools

In order to have a sound basis for classifying SV tools, we characterize each tool by a number of orthogonal attributes.

1. *Nature of the artifact.* The most typical software artifact is code (source or executable), but other kinds of products are also visualized: specification, architectural design, algorithm, data structures, test data, documentation, etc. Some SV tools are restrictive in the sense that they visualize only one type of artifact, whereas others may work for a wide range of products.
2. *Nature of the visualization.* One of the important features of a SV tool is the nature of visualization it provides. The visualization can be static or dynamic.
3. *Notation.* SV tools can be characterized by the form of visualization they provide. Among possible options we mention: graphical (2D or 3D), and textual.
4. *Viewpoint.* The viewpoints can be functional, behavioral, structural or data-centric. The functional viewpoint provides a view of the system in terms of its tasks. The behavioral viewpoint describes the causal links between events and system responses during execution. The data-centric viewpoint is concerned with data objects used within the system and the relationships between them. Finally, the structural viewpoint describes the static relationships between objects (e.g., real-world objects, program entities such as compilation units, sub-program units).

1.2 Classifying SV Tools

SV tools in Software Engineering can be grouped into two broad families. *Construction tools* are used during the requirements, design and coding phases of the development lifecycle. They visualize diagrammatic representations of requirements and designs, and provide visual environments for the programming task. *Analysis tools* are used for code analysis, maintenance and re-engineering. They come into play when non-documented applications must be modified or enhanced, made more maintainable, or migrated to a new platform.

2 Construction Tools

2.1 Requirements and Design Tools

Diagrammatical notations are widely used in software design. For most users, diagrams are usually the most effective way of providing a clear summary of abstract concepts and relationships. However, this is not guaranteed, and a bad diagram can be as useless as a heap of unstructured text. Several visualization tools have been developed to assist designers in their tasks. Most of the tools provide sophisticated graphical editors and user interfaces, but very few offer automatic layout features.

StP/SE [1] integrates various software engineering models into a multi-user environment via seven graphical editors (data flow, data structure, control flow, state transition, control specification, structure chart, flow chart, requirements table). *Statemate* [7] is a graphical tool used to represent reactive system from three perspectives: *structural*, *functional* and *behavioral*. The structural view provides a hierarchical decomposition of the system into its components; the functional view describes the functions and processes of the system; the behavioral view uses statecharts to represent control. The user of Statemate draws statecharts manually, using a graphical editor. *ViSta* [2] *ViSta* is a tool suite designed to support the requirements specification of reactive systems. It enables the user to prepare and analyze a diagrammatic description of requirements using statechart notation. It includes a statechart visualization tool that automatically produces statechart layouts. *Telelogic* [35] is a tool suite used to visualize, develop, implement and test distributed real time system software. It is based on formal languages such as SDL, and graphical notations such as statecharts and UML. The Portable Bookshelf (PBS) [21] is an implementation of the *Software Bookshelf*, a web-based paradigm for the presentation and navigation of information representing large software systems. The PBS Toolkit provides a collection of executables, both stand-alone and web based, which generate and populate a web-based information system for large software systems. PBS provides a default framework for the displaying subsystem landscapes, and links to generated and static documentation for those subsystems. Visual Studio [15] and the individual tools and languages it contains are the foundation for building Windows-based components and applications, creating scripts, developing Web sites and applications, and managing source code.

Most of the OO visualization tools use UML as a visual language to communicate all phases of software development. They provide sophisticated graphical editors to represent: system requirements through *use cases*; component collaboration and scenario analysis through *sequence diagrams*; and architectural modeling through *object modeling*. Additional features include design-level debugging and validation, and code generation. Very few tools provide layout features to optimize the aesthetics of the drawings.

Rational Rose [26] is an award-winning OO model-driven development tool. It allows users to model and develop applications using UML notation. *Rational RealTime* [26] offers notation, processes and tools for real-time design constructs, code generation, and model execution through the UMLTM notation. *Rhapsody* in C, C++ and Java [9] uses the standard UML notation as a basis for design modeling, design validation, and automatic code generation. The *Rhapsody Developer Edition* includes code generation for a variety of operating systems including VxWorks, pSOS, Windows NT, and Windows CE. In addition, it comes with a real-time framework in source code form so users of Rhapsody can easily adapt the generated code for any other operating system, including proprietary in-house systems. *StP/UML* (Software through Pictures) [1] is a graphical, object-oriented modeling environment that supports the analysis and design phase of object-oriented applications. StP/UML facilitates the representation of the application in the Unified Modeling Language via nine graphical editors (use-case, class, sequence, collaboration, state, activity, stereotype, component, deployment editors). *DOME* [8] (the DOmain Modeling Environment) comes with a pre-built set of notations which include UML, Coad-Yourdon OOA, Colbert OOSD, IDEF0, and Petri-Nets. These notations can be used as is or can be extended. The user can develop notations for specific applications, complete with custom visual elements, required interface properties and analysis reports.

2.2 Visual Programming Tools

Visual programming tools provide graphical or iconic elements which can be manipulated by the user in an interactive way according to some specific spacial grammar for program construction [6]. Despite their names, languages such as Visual C++ [13] or Visual J++ [14] are not visual programming languages. They are textual languages which use graphical GUIs to make programming easier. Purely visual languages rely exclusively on visual techniques, and programs written in these languages are never translated to textual-based representations. Examples of purely visual languages include *Prograph* [24], *Periproducer* [18], *Sanscript* [3], *VisPro* [38] and *WiT* [10]. Hybrid visual languages are characterized by the fact that they either involve the use of graphical elements in textual languages, or they translate visual programs to text-based languages at compile time.

2.3 General Purpose Graph Layout Tools

General purpose graph layout systems provide powerful layout tools which automatically calculate the positions of nodes and edges based on specific aesthetic criteria. A layout style emphasizes certain graph characteristics and is used to represent specific applications. For example, hierarchical layouts emphasize precedence relations, and are ideal for representing design notations such as statecharts, structure charts or class diagrams; orthogonal layouts create schematic representations and are adequate for representing entity-relationship diagrams, or object-oriented designs; circular layouts portray interconnected ring and star topologies, and are adequate for representing networks of software agents. Even though these tools offer powerful layout features, they still need to be tailored to specific applications before being used. Among the numerous general purpose graph layout tools, we mention: *Visio* Technical [16], *Tom Sawyer Graph Layout Toolkit & Graph Editor Toolkit* [30], *aiSee* [5], *GraphViz* [12], *GraphEd*, [19], *GraphLet* [20], and *Viztool* [29].

2.4 Characterizing Construction Tools

Construction tools are used in the early stages of software development, hence the artifacts subject to visualization are requirements documents, architectural and detailed designs, and code (undergoing development). The nature of the visualization is static, and the notation is either textual or graphical and two-dimentional. Because of the wide range of notations used in the design phase, the four viewpoints are covered.

3 Analysis Tools

3.1 Program Visualization

Program visualizations vary widely. They can be as simple as indentation of source-code or as elaborate as graphical display of functional dependencies among methods. The goal is to obtain valuable graphical visualizations that do not complicate the comprehension of source unnecessarily. As with all visualizations, graphical types can become unwieldy and, as a result, unused.

xSuds software visualization and analysis toolsuite [34] is a set of software testing, analysis, and understanding tools for C/C++ programs. The tools were developed to analyze the dynamic behavior of software and to allow the user to visualize all program data through an integrated graphical user interface. Many of the tools use color, simplified charts, and textual notations to express the specific visualizations. *Grasp* [37], a Graphical Representations of Algorithms, Structures, and Processes , provides a control flow and data structure diagram that is designed to fit into the space that is normally taken by indentation in source code. The visualization allows the user to recursively fold up a structure (method, loop, if statement, etc.) with a double click of the mouse, then unfold it level-by-level. Syntax based coloring, selection, and find-and-replace are provided. *Source navigator* [32] is a source code analysis tool for C, C++, Java, Tcl, FORTRAN, and COBOL. It displays relationships between classes and functions and members, and displays call trees. *Jtest* [23] is a java testing tool. Jtest automatically performs white-box testing, black-box testing, regression testing, and static analysis (coding standard enforcement) of Java code and presents the results of the tests in graphical format.

3.2 Re-engineering Tools

Re-engineering tools are among the most recent additions to the SV genre. They aid in understanding legacy code and enable users to easily maintain or migrate the system to new platforms. A legacy system that was written years ago by a long-absent programmer, can be re-written by the tools into a new language with new documentation and graphical views. This is not to say that intervention is not required by a software engineer, only that their task is made much simpler.

Understand [28] is used for overall system understanding. Understand parses source code to reverse engineer, automatically document, calculate code metrics. Views include call (invocation) trees, call by trees, generic instantiation trees, with trees, with-by trees, colorized source browsing/editing and detailed HTML/text documentation of analysis information. *Fujaba* using a graphical UI, accomplishes round-trip engineering with UML, SDM, Java and Desgin Patterns. Fujaba combines UML class diagrams, UML behaviour diagrams, SDMstory diagrams, and design patterns for a formal system design and specification language. *Rigi* is focused on the reverse engineering of legacy systems. Inherent in the model is the notion of nested subsystems that encapsulate detail, providing high level overviews of software systems presented using a graphical interface. *Together ControlCenter's* [36] core visualization features include a UML-modeling editor for simultaneous round-trip engineering for multiple languages. Together also generates documentation for Java programs in JavaDoc format. *Discover* [31], provides a complete re-engineering environment for C code. *Imagix 4D* [11] collects data from various types of C and C++ data sources and creates control flow graphs, flow charts, call graphs, class hierarchies, class relationships, etc. It documents files, classes, functions automatically.

3.3 Characterizing Analysis Tools

The main artifact analyzed and visualized by analysis tools is an existing body of source code. The visualization is mostly static, even though some tools offer animation features. The notation is mainly graphical and two-dimensional (3D tools are still in the experimental stage). Code analysis leads to graphical representations that reflect all four viewpoints.

4 Conclusion

Over the years, visualization tools have evolved to meet the increasing demands of the software industry. Yet, as software projects grow to monumental size and complexity, the need for a new generation of tools is more pressing than ever. Most of the existing construction tools offer nothing more but sophisticated graphical editors for visualization. While these tools help in the analysis and design of small to medium software projects, they are useless for large, complex applications. In this case, it is imperative to complement them with powerful layout features, such as the ones developed by the graph drawing community (see section 2.3). In addition, very few construction tools offer dynamic visualizations, i.e., animation and simulation. It is apparent that more research is needed in this area. Finally, even though some visual programming environments are available on the market, these tools are still not widely used by the software engineering community. It may be interesting to assess these tools and determine the features that will make them more appealing to software engineers.

With respect to analysis tools, it seems that most of the work has concentrated on the development of elaborate information extraction methods. Most of the analysis tools rely on general purpose graph layout tools to visualize information. Because these tools are generic, they do not take into account the drawing conventions of accepted software engineering notations (e.g., UML, data flow diagrams, etc). Additionally, these layout tools do not scale. In re-engineering, software artifacts may necessitate the visualization of graphs consisting of tens of thousands of nodes. Hence, in addition to improving the existing layout algorithms, it is necessary to develop new navigation mechanisms to browse through complex representations of code. Or better, it may be the time to investigate new visualization techniques.

This chapter starts with a paper by Rainer Koschke from the University of Stuttgart discussing visualization tools for reverse engineering. The author describes *Bauhaus*, a system which recovers architectural designs from C programs. The visualization of the recovered information is produced by the generic graph editor of *Rigi* [22], and the general purpose graph layout tools *Graphed* [19] and *Graphlet* [20].

Wim De Pauw et al. from IBM Watson Research Center describe Jinsight, a tool that visualizes the run-time behavior of Java programs. Jinsight allows the user to explore program execution through several linked views (e.g., histogram view, reference pattern view, execution view, etc.).

Katherina Mehner from the University of Paderborn discusses a visualization and debugging environment for concurrent Java programs. The UML-based visualization is achieved by integrating the tool with *Together* [36].

Rainer Oechsle and Thomas Schmitt from the Universities of Trier and Saarbrücken describe *JAVAVIS*, a system used to visualize the dynamic behavior of sequential JAVA programs. The tool displays the object and sequence diagrams of a running program.

Thomas Zimmermann and Andreas Zeller from the Univerity of Passau discuss the visualization of memory graphs. A memory graph is a construct that captures the state of a running program at specific times during its execution. This paper discusses an analysis and extraction method. The visualization is produced by the generic graph layout tool *Graphviz* [12].

References

1. Aonix. Software through pictures. Downloaded. `http://www.aonix.com/content/index.html`.
2. R. Castello, R. Mili, and I.G. Tollis. Vista: A tool suite for the visualization of statecharts. *Journal of Systems and Software*. To appear, 2002.

3. Northwoods Software Corporation. Sanscript. Downloaded. `http://www.nwoods.com/sanscript/index.htm`.
4. J. Domingue, B. A. Price, and M. Eisenstadt. A framework for describing and implementing software visualization systems. In *Proceedings of Graphics Interface '92*, pages 53–60, Vancouver, Canada, May 1992. available at `ftp://watson.open.ac.uk/pub/documents/Domingue-Viz-GI92.ps.Z`.
5. AbsInt Angewandte Informatik GmbH. aisee. Downloaded. `http://www.absint.de/aisee.html`.
6. E. J. Golin. A method for thespecification and parsing of visual languages. Ph.D. dissertation. Brown University, 1990.
7. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, May 1990.
8. Honeywell. Dome. Downloaded. `http://www.htc.honeywell.com/dome/`.
9. I-Logix. Rhapsody. Downloaded. `http://www.ilogix.com/frame_html.cfm`.
10. Coreco Imaging. Wit. Downloaded. `http://www.logicalvision.com/default.htm`.
11. Imagix. Imagix 4d. Downloaded. `http://www.imagix.com/products/imagix4d.html`.
12. AT&T Labs-Research. Graphviz. Downloaded. `http://www.research.att.com/sw/tools/graphviz/`.
13. Microsoft. Visual c++. Downloaded. `http://msdn.microsoft.com/visualc/`.
14. Microsoft. Visual j++. Downloaded. `http://msdn.microsoft.com/visualj/`.
15. Microsoft. Visual studio. Downloaded. `http://msdn.microsoft.com/vstudio/`.
16. Visio Corporation Microsoft. Visio technical. Downloaded.
17. B.A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
18. Nortel Networks. Periproducer. Downloaded. `http://www.peri.com/product/periprbr.html`.
19. University of Passau. Graphed. Downloaded. `http://www.infosun.fmi.uni-passau.de/GraphEd/`.
20. University of Passau. Graphlet. Downloaded. `http://www.infosun.fmi.uni-passau.de/Graphlet/`.
21. University of Toronto. Pbs, portable bookshelf. Downloaded. `http://swag.uwaterloo.ca/pbs/`.
22. University of Victoria. Rigi: A visual tool for understanding legacy systems. Downloaded. `http://www.rigi.csc.uvic.ca/`.
23. ParaSoft. Jtest. Downloaded. `http://www.parasoft.com/products/jtest/index.htm`.
24. Pictorius. Prograph. Downloaded. `http://www.pictorius.com/prograph.html`.
25. B.A. Price, R.M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.
26. Rational. Rational rose. Downloaded. `http://www.rational.com/products/rose/index.jsp`.
27. G. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computers*, pages 97–123, December 1993.
28. Inc. Scientific Toolworks. Understand for ada. Downloaded. `http://www.scitools.com/uada.html`.

29. J. Six and I.G. Tollis. A framework for circular drawings of networks. In *Proceedings 1999 International Symposium on Graph Drawing (GD '99), Lecture Notes in Computer Science*, volume Vol. 1731, pages 107–116. Springer-Verlag, 1999.
30. Tom Sawyer Software. Tom sawyer graph layout toolkit & graph editor toolkit. Downloaded. `http://www.tomsawyer.com/products.html`.
31. Upspring Software. Discover. Downloaded. `http://www.upspringsoftware.com/products/discover/index.html`.
32. Cygnus Solutions. Source navigator. Downloaded. `http://sources.redhat.com/sourcenav/`.
33. J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT press, 1998.
34. Telcordia Technologies. xsuds, software visualization and analysis toolsuite. Downloaded. `http://xsuds.argreenhouse.com/`.
35. Telelogic. Telelogic doors. Downloaded. `http://www.Telelogic.se/`.
36. TogetherSoft. Together/j. Downloaded. `http://www.togethersoft.com/products/controlcenter/`.
37. Auburn University. Grasp: Graphical representations of algorithms, structures, and processes. Downloaded. `http://www.eng.auburn.edu/department/cse/research/grasp/`.
38. K. Zhang, D-Q. Zhang, and J. Cao. Design, construction, and application of a generic visual language generation environment. *IEEE Transactions on Software Engineering*, 27(4), April 2001.

Software Visualization for Reverse Engineering

Rainer Koschke

University of Stuttgart,
Breitwiesenstr. 20-22,
70565 Stuttgart, Germany
koschke@informatik.uni-stuttgart.de,
http://www.informatik.uni-stuttgart/ifi/ps/rainer

Abstract.

This article describes the Bauhaus tool suite as a concrete example for software visualization in reverse engineering, re-engineering, and software maintenance. Results from a recent survey on software visualization in these domains are reported. According to this survey, Bauhaus can indeed be considered a typical representative of these domains regarding the way software artifacts are visualized. Specific requirements for software visualizations are drawn from both the specific example and the survey.

1 Introduction

Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships and create representation of the system in another form or at a higher level of abstraction, whereas *re-engineering* is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring [Chikofsky & Cross, 1990].

Research in reverse engineering focuses on extracting, storing, presenting, and browsing information, reducing the amount of unnecessary information for a particular task, analyzing the extracted data and to build useful abstractions of the system under analysis. Because reverse engineering is generally a highly interactive and incremental process, in which results of automatic analyses need to be presented to the reverse engineer that are then validated, augmented, and fed back to following automatic analyses, software visualization plays a key role in reverse engineering. Presenting the data to the reverse engineer in a suitable manner is a main issue here and the reverse engineering research community struggles with finding solutions to this problem.

In the next section, the Bauhaus tool suite is described as one concrete example of software visualization for reverse engineering. The example is not a particularly advanced use of software visualization. As a matter of fact, I rather believe that there is still enough room for improvements. The example was chosen because it can be considered typical for the domain according to a survey that I recently conducted [Koschke, 2001]. Bauhaus is rather an example for the state of the practice than for the state of the art with respect to its software visualization capability. With respect to its analytical reverse engineering capabilities, it is more advanced. Yet, its exact analyses are beyond the scope of this article. The results of the more general survey are presented in Section 3.

2 Architecture Recovery in Bauhaus

The *Bauhaus project* researches reverse engineering techniques to help program understanding of legacy code [Bauhaus, 2001]. Bauhaus has support for frequent maintenance tasks that involve program-understanding-in-the-small (points-to and side effect analyses, detection of uses of uninitialized variables and dead code, program slicing, etc.) and re-engineering tasks that require knowledge of the system's architecture and hence are more oriented toward program-understanding-in-the-large.

This section describes the software visualization in Bauhaus. However, we will start with some background information on reverse engineering.

2.1 Reverse Engineering Background

Analyzing a system, we can roughly distinguish three different levels of abstraction:

- The *lower level* represents the source code in a way that contains *all necessary details* of syntactic, semantic, control and data flow information.
- The *middle level* only contains *global* information that can be automatically extracted from source code, like global variables, functions, user-defined types and their relationships. The middle level is the seam between the lower level and the next upper level, namely, the architectural level.
- The *architectural level* contains architectural information.

Bauhaus seeks to recover architectural descriptions from source code. According to the *IEEE Standard on Recommended Practice for Architectural Description of Software-Intensive Systems* [IEEE-Std-1471-2000], an *architectural description* is a collection of products to document an architecture. An *architecture* is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. A *view* is a representation of a whole system from the perspective of a related set of concerns. A *viewpoint* is a specification of the conventions for constructing and using a view. A viewpoint acts as a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [IEEE-Std-1471-2000].

2.2 Extracted Artifacts

Information about the system is exclusively extracted from the source code in Bauhaus – because this is the only reliable source of information – and represented at each level as graphs with varying granularity. The maintenance analyses are based on information at the lower level only and are fully automatic. Their results are generally represented as links to the original source code. On the other hand, architecture recovery with Bauhaus is a semi-automatic process that involves the Bauhaus user (presumably a software maintainer or auditor). Due to the highly interactive nature of this process, software visualization is an important part. While Bauhaus uses marked-up text and hypertext for the results of the more source-code oriented maintenance analyses at the lower level, the architecture recovery tasks have higher demands on software visualization.

At the middle level, information extracted from the source code is represented as a resource graph (RG) [Koschke, 2000]. The resource graph is a coarse-grained intermediate representation that represents concrete as well as conceptual information in form of entities and their relationships. Concrete information constitutes a *base view* and is global information that can be directly extracted from the source code, yet abstracts from a particular source language, such as call, type, and use relations. The conceptual information is added by the Bauhaus user based on results from automatic reverse engineering analyses. Currently, the Bauhaus toolkit offers techniques for component and connector recovery. In the following, we will focus on the component recovery part of Bauhaus. The semi-automatic process of component recovery will be described in Section 2.4.

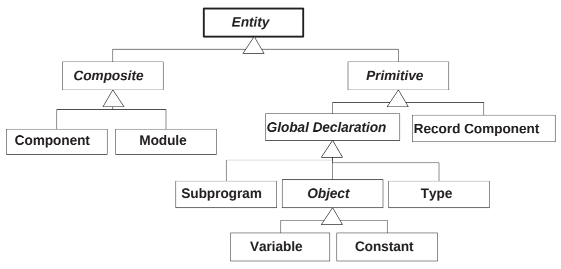


Fig. 1. Entities of the Resource Graph

The entities of the RG are programming language constructs and recovered abstract concepts. The entities, which are represented as nodes in the RG, are shown by the UML inheritance model in Figure 1. Relationships are represented as edges in the RG. Examples of relationships range from information that can be directly extracted from the source code (e.g., function calls, variable references, function signatures) – as shown in Figure 2 as *base viewpoint* – to more abstract concepts (e.g., the *conceptual component decomposition* as shown in Figure 2). The RG can be visualized and manipulated in Bauhaus with our extension of the graph editor Rigi [Müller, 1992]. The RG is the shared knowledge base – so-to-speak – between automatic analyses and the Bauhaus user.

Even though the RG is a coarse-grained representation of the system, the RG gets large for large systems. Figure 3 shows the number of nodes and edges in the RG that represent extracted information for five C systems (Concepts, Aero, Bash, CVS, and Mosaic) in the range of 7 to 52 KLOC (thousand lines of code). Interestingly enough, as Figure 3 shows, the RG size is not necessarily a monotonic function of the program size in terms of commented lines of codes (Figure 3 only contains the entities and relationships of the base viewpoint as described in Figure 2). The RG for the system Aero, which has about 28 KLOC, has more nodes and edges than the RG for the three larger systems (except for the number of edges for the largest system, namely Mosaic). The number of nodes per lines of code depends on the programming style, in particular the number of lines of code per subprogram and the use of global variables. However, the graph density in terms of number of edges per node seems to be a constant factor of about four edges per node at the middle level.

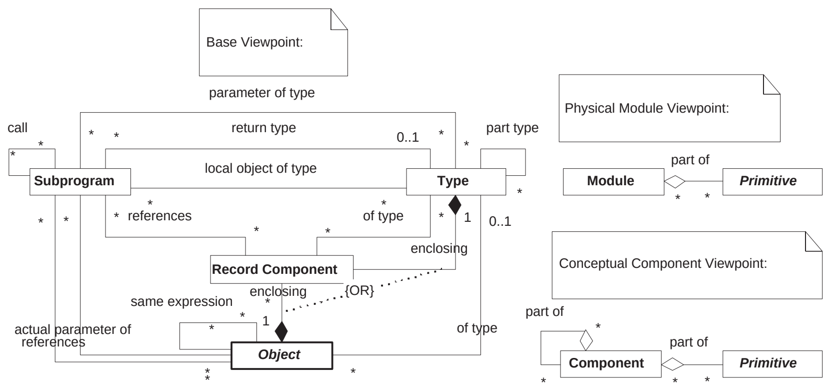


Fig. 2. Base, Physical Module, and Conceptual Component Viewpoints

2.3 Recovered Conceptual Artifacts

One particularly developed part of Bauhaus supports component recovery for procedural programming languages, specifically for C. *Components* are cohesive groupings of related global declarations of subprograms, variables, and user-defined types. The decomposition of the system into components is a structural viewpoint of the conceptual architecture. Component views are needed for re-modularization, migration to object-oriented languages, and program understanding in general. The component view gives a maintainer a conceptual perspective of the system and is in contrast to the *physical module view* that shows how the system is decomposed into physical modules (or files in the programming language C). The latter view can trivially be derived by looking at the existing files that make up the system. However, the physical module view shows the system as built and may not necessarily reflect how the decomposition of the system should be decomposed into cohesive groupings. Due to ad-hoc maintenance tasks the (originally well-designed) physical module decomposition may be deteriorated, i.e., show high coupling between modules and low cohesion within modules. Conceptual components, by definition, are always cohesive.

Figure 2 shows the description of the *physical module view* consisting of modules, program primitives, and their part-of relationships and the *conceptual component view* consisting of modules, program primitives, and their part-of relationships. Note that components can be hierarchical whereas modules that model C header (include files with suffix *.h*) and body files (that contain function definitions with suffix *.c*) cannot contain other modules.

The process of recovering the component view is described in the following section.

2.4 Component Recovery Process

Bauhaus supports an iterative semi-automatic method to detect *components*, also known as *logical modules* or *objects*. The analysis cycle consists of the following steps (see Figure 4) [Koschke, 2000]:

1. The Bauhaus user selects one ore more fully automatic techniques. Currently, 15 different techniques are available, many of them have additional variants. The description of the techniques is beyond the scope of this article. A detailed description can be found in [Koschke, 2000].

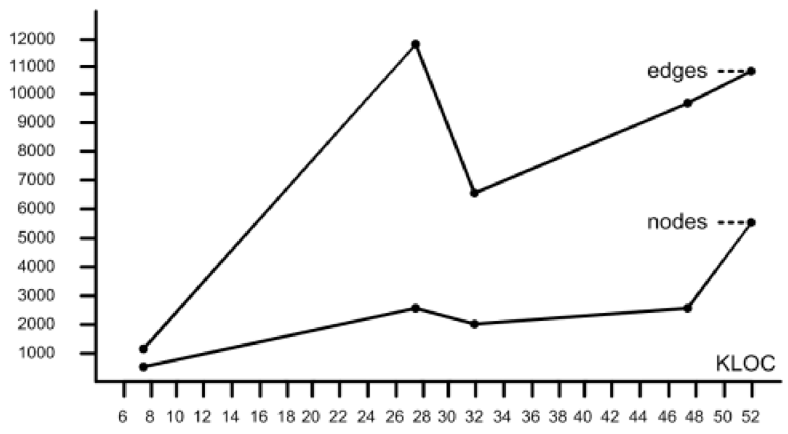


Fig. 3. Number of Nodes and Edges at the Middle Level

2. The selected techniques use as input the base view that contains the program primitives and their relationships extracted from the system (see Figure 2) and the so-called *user view* that contains the components that have been found so far. In the beginning, the user view is empty.
3. Each analysis application yields one component view, which can be assessed by certain metrics (e.g., number of elements, name similarities, cohesion, etc.). Multiple component views of different analyses may be combined by intersection, union, and difference based on fuzzy sets.
4. The resulting combined component view may then be validated by the user. The user can reject components in part and as a whole and may add additional entities to existing components. Accepted components are moved to the user view.

The user controls the detection process by selecting analyses and metrics and by validating the candidates proposed by the automatic techniques. The task of the computer comprises the automatic analyses, computation of the metrics for the proposed candidates, presentation of the results, and bookkeeping of the user decisions.

An analysis selected by the user takes into consideration the components that were previously confirmed by the user (in the first iteration there are none). Thus, the analyses are applied incrementally. In each iteration, the user selects and combines different analyses to find components that could not be found by previous analyses. The process ends when the found components are sufficient for the task at hand or no further component can be found anymore. Each intermediate and resulting view can be visualized as a graph as described in the following section.

2.5 Software Visualization in Bauhaus

Suitable visualization is an important issue in the above process of architecture recovery. The user should be able to quickly grasp the represented information. Moreover, he or she should also be able to quickly derive other information that might be needed, yet not forseen by our analyses. Hence, beyond pure visualization, browsing capabilities for the large information space need to be offered, too. Since our research focus in on the analytic parts of architecture recovery rather than visualization or browsing of architectural views and developing an own visualization and browsing tool is a major effort, we used and extended the generic graph editor, Rigi, originally developed by Hausi Müller and his team at the university of Victoria in Canada [Müller, 1992].

Rigi is an interactive, visual graph editor designed to help to better understand and redocument software. The underlying multi-graph consists of typed attributed nodes and directed edges. Special level edges allow for hierarchical graphs. Rigi provides selection, filtering, and editing operations, dependency and change impact reports, overview and projection perspectives, metrics for cohesion and coupling, views to capture interesting perspectives, a scripting language and command library, annotations of nodes and edges, and a customizable

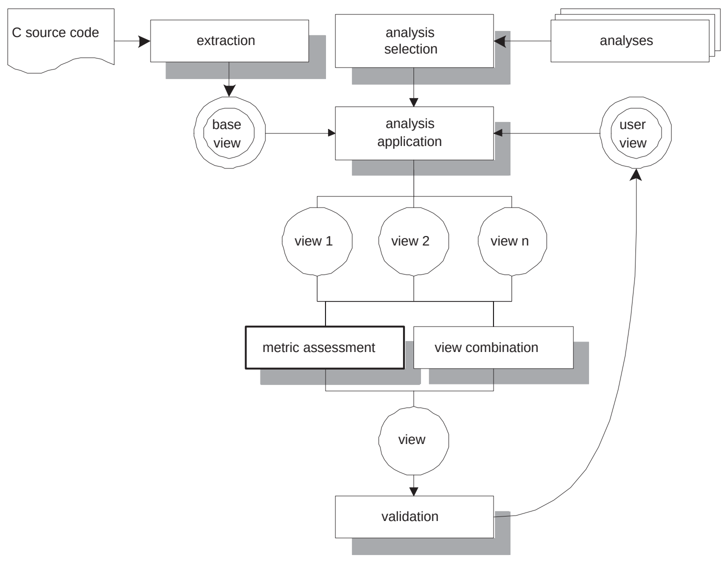
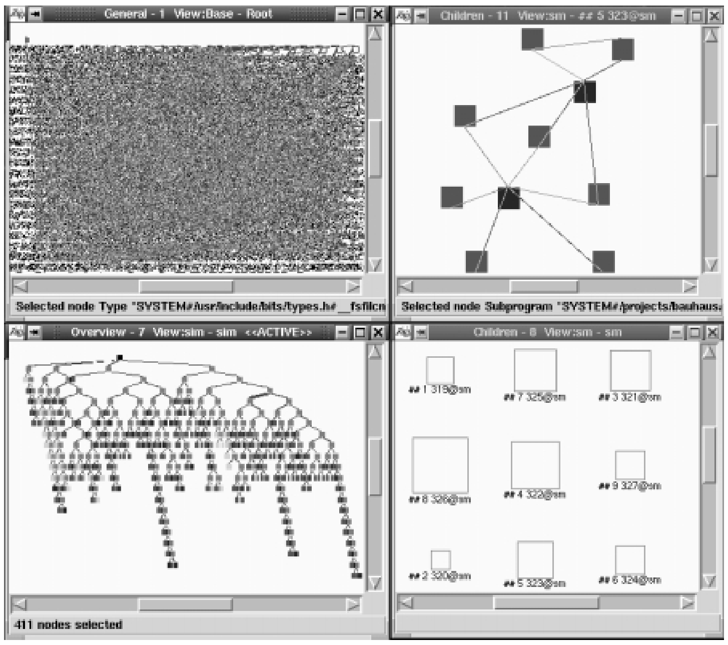


Fig. 4. Process of Component Recovery in Bauhaus

user interface. Nodes may be linked to source positions and a user-defined text viewer may be opened to show the source text when the user clicks on the node. Rigi is adaptable to different languages and purposes.

Rigi also offers different automatic graph layouts. Only some of them are built-in layouts, namely, those for trees and grids. For more advanced layouts, `Graphed` is used as an external layouter for spring embedder and Sugiyama's layout [GraphEd]. Our group has integrated `Graphlet` as an additional external layouter [Graphlet]. Tree layouts are used in Bauhaus/Rigi for the dominance tree, which shows local functions in the call graph and subsystems in the dependency graph, and for the result of hierarchical clustering techniques for component recovery. Spring embedder and Sugiyama's algorithm are used for call graphs and type and reference relationships.

Figure 5 shows a few visualization examples with Bauhaus/Rigi. The left upper window shows all nodes and edges of the base view (the base view is described in Figure 2) for a 7.5 KLOC system, overlapping each other. Obviously, this visualization is useless. In order to understand all the relationships that are there, filtering and selection mechanisms are needed. The upper right window shows the contents of a component that contains two types and eight subprograms. The visualization immediately shows that three subprograms are connected to both types and all others are connected to one type only. The lower left window shows the result of a hierarchical clustering technique. The iterative hierarchical clustering technique groups in each step the two most similar subtrees based on a similarity metric in a bottom-up fashion. The visualization shows the order of this clustering, i.e., the most similar entities can be found in lower subtrees. The higher a subtree, the less similar are its elements. Each inner node (which represents a grouping) is annotated with its similarity value. This visualization is an important guidance for a user's validation. The user can start at the leaves and then climb up the tree until a grouping gets doubtful. In principal, the same information could also be "visualized" as a matrix of similarity values, but the superiority of the visualization as tree is obvious. The visualization is even active in the sense that a user can reject and accept groupings within the visualization and obtain the source code of contained programming primitives by one mouse click. The lower right window shows an example on how components can be assessed with metrics. The metric value is expressed as the size of the nodes that represent components. This way the relation between the components with respect to the given metric is easy to grasp.


General - 1 View:Base - Root
Children - 11 View:sm - ## 5 323@sm
Overview - 7 View:sim - sim <<ACTIVE>>
Children - 8 View:sm - sm
## 1 319@sm
## 7 325@sm
## 3 321@sm
## 8 326@sm
## 4 322@sm
## 9 327@sm
## 2 320@sm
## 5 323@sm
## 6 324@sm

Selected node Type "SYSTEM#/usr/include/bits/types.h#__fsfilcn

Selected node Subprogram "SYSTEM#/projects/bauhaus.

411 nodes selected

Fig. 5. Different Kinds of Visualization with Bauhaus/Rigi

The Bauhaus GUI, which is based on Rigi, has a few unpleasant properties. It is relatively slow, which can cause noticeable waiting periods for large graphs and hence sometimes disrupts the fluent use of the tool. Graphs with more than 500 nodes, graphs with multiple edges between the same nodes, and disconnected graphs cannot be automatically layouted due to limitations of the external layouter `Graphed`. `Graphlet` does not have these principal restrictions, but it is much slower than `Graphed` and its layouts cannot be used for larger graphs in an interactive application.

The visualization of nodes and edges is limited, too. All nodes have the same shape of a rectangle. Semantics of nodes and edges is encoded by colors only, making it difficult to distinguish nodes and edges if one has many different types. Nodes have only two ports to which edges can be connected – one for incoming, one for outgoing edges. If two nodes have multiple shared edges, edges lay on top of each other. Edges are only straight lines and may not have bends, which basically makes use of planar graph layouts impossible.

3 Software Visualization for Reverse Engineering in General

Software visualization in Bauhaus is rather typical for the domain of reverse engineering as can be seen by a survey recently conducted [Koschke, 2001]. This section describes the findings of the survey.

The survey was conducted by way of a questionnaire sent via email to researchers in the areas of software maintenance, reverse engineering, and re-engineering. The list of researchers was compiled from several lists of attendees and PC members of conferences related to these fields, namely, the *International Conference on Software Maintenance* (ICSM), the *Working Conference on Reverse Engineering* (WCRE), the *International Workshop on Program Comprehension* (IWPC), and the *European Conference on Software Maintenance and Re-engineering* (CSMR). The list contained about 580 different email addresses. The response rate of the survey was about 20 percent. Out of the 111 answers, 83 researchers confirmed that they are active researchers in the area of software maintenance (absolute number: 56), reverse engineering (52), re-engineering (36), metrics (25), and related domains (6), where multiple selections were possible.

Roman and Cox define *program visualization* as the mapping from programs to graphical representations [Roman & Cox, 1992]. The definition is general enough that we can widen it to other kinds of software artifacts (including programs). Consequently, we define *software visualization* as the mapping from software to graphical representations. Different categories of software visualization may be distinguished according to the following criteria with respect to this mapping [Roman & Cox, 1992]:

- Scope: what aspect of the software is visualized?
- Abstraction: what kind of information is conveyed by the visualization?
- Specification Method: how is the visualization constructed?
- Technique: how is the graphical representation used to convey information?

The survey on software visualization in the area of software maintenance, reverse engineering, and re-engineering has shown that the scope of the visualization is quite diverse [Koschke, 2001]. It ranges from module and subsystem dependencies, call graphs, object models, software architectures, web artifacts, semantic nets and ontologies, control and data flow, database schemas, directory structures to source text.

In terms of Roman and Cox's taxonomy, the abstraction mechanism embodied in common reverse engineering tools is usually a structural representation that is obtained by concealing or encapsulating some of the details associated with the software or its execution and using a direct representation of the remaining information [Roman & Cox, 1992]. Graphs are typically used to depict program structures, dependencies, control and data flow. The information presented to the viewer is present in the program, although simply obscured by details. The representation simply conveys the information in a more economical way by suppressing aspects not relevant to the viewer. For instance, a call graph shows aspects of the software's global control structure, but at the same time it suppresses detailed aspects of the call sites, like the conditions that must hold for the calls to happen and the exact number and order of multiple calls in the same function body.

The specification method is generally predefined, i.e., the viewer cannot really influence the way the information is presented. Many systems allow the viewer to specify colors or shapes for the visualization or select different kinds of automatic graph layouts, but the principle way of visualizing is generally fixed.

The techniques most maintenance, reverse engineering, and re-engineering tools use to visualize information are centered around graphs and text, as shown by the recent survey [Koschke, 2001]. Among the selected representations, graphs are used in 52% of the cases (many of them are hierarchical graphs). In 18% of the cases, UML diagrams are used. Text and hypertext are used in 18% of the cases, but we may assume that a textual representation is actually much more often used than responded – many people do not perceive text as a way of software visualization. Other, less frequently used ways of representation are scatter plots, charts, process flows, database models, and tables.

Animation is rarely used. Only 12.5% of the respondents said that they are using animated representations. Interestingly enough, 40% do believe that animation is useful (in particular for dynamic information) and 34% responded that it might be useful – whereas only 15% believe it is not useful at all (11% did not answer the question).

Since graphs are the dominant way of visualization, the question is raised whether automatic graph layout algorithms are used. As a matter of fact, 71% of the respondents use automatic graph layouters (12% did not answer the question). Among these, surprisingly many have implemented their own graph layout algorithms (28%). Only 41% use the readily available non-commercial or commercial graph layout packages (31% did not answer the question). The most frequently used layout package for graphs is the `GraphViz` system by AT&T (12 in absolute numbers) [GraphViz]. `GraphEd` [GraphEd] and its successor, `Graphlet` [Graphlet], together amount to 7 users. Another 9 use commercial UML tools for rendering UML models, like Rational Rose or Together. `VCG` is used by 5 [VCG], whereas the commercial layout package by Tom Sawyer Software is used by 2. Three people are using Rigi (these people also use the integrated `GraphEd` implicitly; the number of users given for `GraphEd` includes the Rigi users). The remaining 10 people use other packages such as Java2D, Java3D, Microsoft Visio, daVinci, and others.

The most frequently used class of layout algorithms is those for trees (10 in absolute numbers); 9 people use a Sugiyama algorithm, and 7 a spring embedder. Only 2 people use planar graph layout algorithms. The remaining 8 use other, less known layouts like Tunkelang, Minbackward, Barymedian, Manhattan Edges, X-Dags, SequoiaView, and others.

The encouraging message for researchers in the area of software visualization is that 40% of the interviewed people believe software visualization is absolutely necessary for software maintenance, reverse engineering, and re-engineering and still 42% think software visualization is important but not critical. 7% think that it is at least relevant and 6% that they can do without but it is nice to have. Only 1% (actually, a single person) believes software visualization is not an issue at all (4% did not answer the question).

Even though most people acknowledge the importance of software visualization, relatively few people consider it their primary research (11%) or at least a substantial part of their research (18%). Many people are doing research in software visualization every now and then (20%). The relative majority is primarily using or integrating existing software visualization tools developed by others (33%). 11% do not deal with software visualization at all. 7% did not answer the question.

4 Conclusion

We conclude with listing problems of software visualization for software maintenance, reverse engineering, and re-engineering drawn from our own experience in Bauhaus and the survey. I do not think that they are all specific to the mentioned domains, but will arise for all domains in which large and semantically rich information spaces are to be visualized.

Semantics. Graphs are frequently used to represent information. However, these graphs do have semantics and automatic layouts should take the semantics of nodes and edges into account and also the conventions used to draw such graphs manually. For instance, in a UML class model, one would expect to direct all inheritance relationships in one direction; all other kinds of relationships are subordinated.

Size. The amount of data that need to be visualized can be rather large; graphs with 4,000 nodes and more are typical. One may argue that graphs at this size should not be visualized at once since they cannot be understood at this size anyway. In fact, one needs additional navigation, selection, and filtering mechanisms. However, even if a larger call graph with more than 1,000 nodes (still a small system) should be presented in excerpts, the layout for the whole graph needs to be computed in advance. Then a "lense perspective" could be used to browse the large graph showing only subgraphs – where the nodes in the whole graph as well as in the mental map of the viewer would keep their position while the viewer moves the lense.

Evolution. Maintenance and reverse engineering activities require weeks, months, or even years, and usually one cannot afford to freeze normal development. Consequently, the system is permanently under change and, hence, there is not just one graph, but many graphs that are derived from each other. Visualizations may evolve and one has to keep track of this evolution

Multiple users. Large maintenance and reverse engineering projects require team-work and, hence, visualizations need to support multiple users that may work at the same system at the same time at possibly different locations.

Multiple views. Maintenance and reverse engineering involve different stakeholders and, thus, require multiple perspectives from which a system may be viewed. Moreover, different dimensions of the data need to be visualized, like the time dimension or level of abstraction. Multiple views raise the questions of how to integrate these views, how to navigate within and between views, and how to preserve the context during navigation?

Static and dynamic visualization. Most kinds of visualization in maintenance, reverse engineering, and re-engineering are static. However, for dynamic aspects animated visualization would be useful.

Cognitive models. There is a lack of cognitive models for visual understanding and empirical evidence for appropriate visualizations, i.e., we currently do not really know how maintainers grasp visualized artifacts and which kinds of visualization work best for a specific problem. If we knew answers to the latter question, we could also try to automatically select the right kind of visualization depending upon criteria of the input data to be visualized.

Interoperability. No single tool alone can solve the manifold and complex problems of reverse engineering. Consequently, several tools need to be integrated, which requires a high degree of interoperability among tools. Currently, the reverse engineering community works on interoperability issues, in particular, on data exchange and standard schemas. GXL has been evolved to a standard vehicle for data exchange among reverse engineering research tools [GXL]. GXL basically allows one to transfer graphs. It would be advantageous to the reverse engineering and software visualization community to agree upon a joint exchange format.

The Dagstuhl seminar on software visualization has brought together many researchers from very different areas of software visualization. Most of their ideas on software visualization are specifically interesting for the domain of maintenance, reverse engineering, and re-engineering and may help to overcome some of the problems mentioned above. Strangely enough, there is surprisingly little overlap between the communities for reverse engineering and software visualization in terms of people despite of the large overlap in terms of topics. It is high time for our communities to team up since our common goal is to help programmers understand programs.

References

[Bauhaus, 2001] Bauhaus, http://www.bauhaus-stuttgart.de.

[Chikofsky & Cross, 1990] Chikofsky, E.J.; Cross II, J. H.: Reverse Engineering and Design Recovery. IEEE Software, pp. 13-17, January, 1990.

[Graphlet] Graphlet, http://www.graphlet.de.

[GraphViz] GraphViz, http://www.research.att.com/sw/tools/graphviz

[GXL] GXL, http://www.gupro.de/GXL. See also the paper by Andreas Winter et al. in these proceedings.

[GraphEd] GraphEd, http://www.uni-passau.de/GraphEd

[IEEE-Std-1471-2000] IEEE-Std-1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE, 2000.

[Koschke, 2000] Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. Dissertation, Institute of Computer Science, University of Stuttgart, Germany, 2000.

[Koschke, 2001] Koschke, R.: Survey on Software Visualization for Software Maintenance, Re-Engineering, and Reverse Engineering, http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/softviz

[Müller, 1992] Müller, H.; Wong, K.; Tilley, S.: A Reverse Engineering Environment Spatial and Visual Software Interconnection Models. Proc. of ACM SIGSOFT Symposium Software Development Environments, pp. 88-98, December, 1992.

[Roman & Cox, 1992] Roman, G.-C.; Cox, K. C.: Program Visualization: The Art of Mapping Programs to Pictures. Proc. of the International Conference on Software Engineering, Association of Computing Machinery, 1992.

[VCG] VCG, http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html

Visualizing the Execution of Java Programs

Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang

IBM T.J. Watson Research Center
30 Saw Mill River Road, Route 9A
Hawthorne, NY 10532 USA
`{wim,erikj,nickm,sevitsky,vlis,jeaha}@us.ibm.com`

1 Introduction

Jinsight is a tool for exploring a program's run-time behavior visually. It is helpful for performance analysis, debugging, and any task in which you need to better understand what your Java program is really doing.

Jinsight is designed specifically with object-oriented and multithreaded programs in mind. It exposes many facets of program behavior that elude conventional tools. It reveals object lifetimes and communication, and attendant performance bottlenecks. It shows thread interactions, deadlocks, and garbage collector activity. It can also help you find and fix memory leaks, which remain a hazard despite garbage collection.

A user explores program execution through one or more *views*. Jinsight offers several types of views, each geared toward distinct aspects of object-oriented and multithreaded program behavior. The user has several different perspectives from which to discern performance problems, unexpected behavior, or bugs small and large. Moreover, the views are linked to each other in many ways, allowing navigation from one view to another. Navigation makes the collection of views far more powerful than the sum of their individual strengths.

2 Object Population in the Histogram View

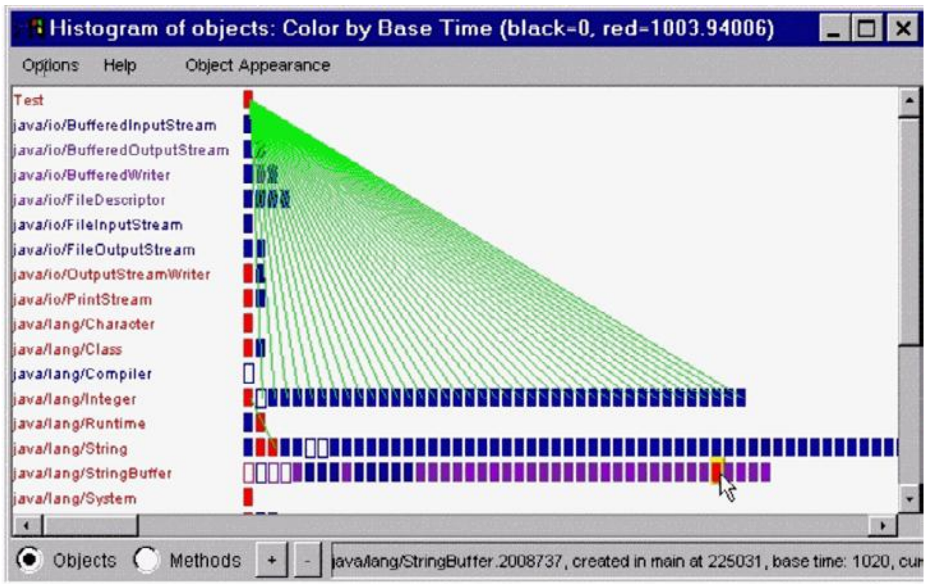
The Histogram view is a basic visualization of resource consumption (CPU and memory) in terms of classes, instances, and methods. It gives an overview of hot spots in a program's execution.

The Histogram view arranges information by class. In Fig. 1, each row shows the name of a class followed by colored rectangles representing the instances of that class. Colors for instances and classes can depict various criteria:

- time spent in methods relating to that instance or class
- number of calls
- amount of memory consumed
- number of threads in which the instance or class participates

Hot spots and patterns of resource usage are immediately visible in this view. You can get more detailed information about individual elements by clicking on them or merely positioning the mouse cursor over them. The view also gives you a good idea of when objects are created and garbage collected. (A rectangle turns into an outline when the object has been collected.) This can help you spot memory leaks, as we explain later.

The lines in the view represent relationships among objects. For example, Fig. 1 shows all the method calls on objects of class java/lang/Integer. You can tell the view to indicate how each object calls, creates, or refers to other objects. Seeing connections among objects is useful for detailed investigation of calling and reference relationships. However, the combinatorial nature of these relationships will make any program larger than “Hello world” hard to examine.



java/lang/StringBuffer.2008737, created in main at 225031, base time: 1020, cur

Fig. 1. Histogram view

3 Pattern Extraction in the Reference Pattern View

Complexity is indeed a challenge when visualizing the execution of object-oriented programs. We deal with complexity in a number of ways. First, a good visualization makes it much easier to interpret complicated behavior than poring through textual data. In this section we discuss another technique, *pattern extraction*, that can simplify visualizations by eliminating extraneous detail. Later in this paper we discuss yet another approach to handling complexity, employing database techniques to structure the information.

Instead of displaying every nuance of the execution, the pattern extractor analyzes execution data for recurrences in the calling and reference relationships. Visualizations can display consolidations of these recurrences, thereby revealing the essential behavior.

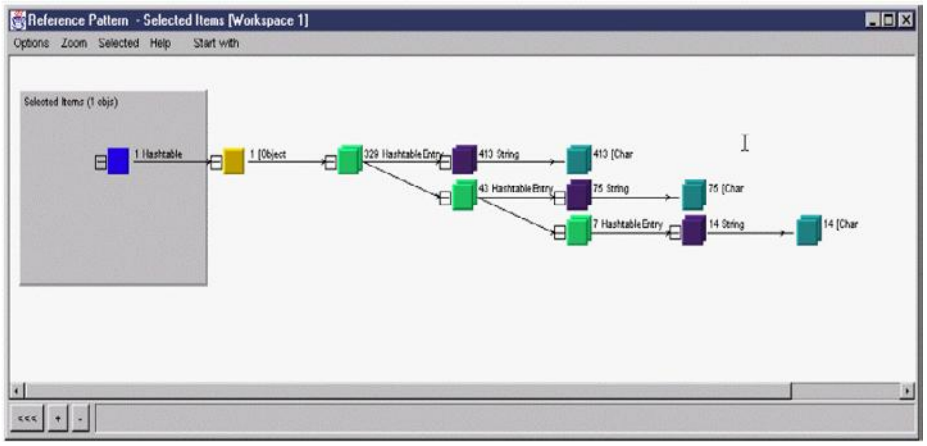


Fig. 2. Reference Pattern view

Fig. 2 shows the result of selecting a Hashtable object in the Histogram view and then examining its current structure in the Reference Pattern view. Instead of showing individual instances, the Reference Pattern view groups them by type and reference, twin squares denoting collections of objects. In the figure we see a square representing the Hashtable object (far left) that points to an array of Object instances—more precisely, 329 HashtableEntry objects. These objects contain 413 references to String objects, which in turn refer to 413 arrays of characters. The original 329 HashtableEntry objects themselves refer to 43 others containing references to yet seven others, along with 75 String instances—and so on. The complete data structure contains more than 1000 elements, making it difficult to visualize when fully expanded. Visualizing the pattern of references lets you view and understand this data structure efficiently.

4 Memory Leak Analysis

We use the same pattern extraction technique to detect memory leaks. The process for finding most memory leaks assumes a simple but common scenario. A user-level operation (for example, the display of a dialog box) creates temporary objects. When the operation completes (the dialog box is closed), we expect all of the temporary objects to be released—but some are not.

Fig. 3 illustrates this scenario schematically. The program has reached a stable state; its object population is shown in the lower area. The user of the program performs an operation that creates temporaries, which appear in the upper area. When the operation terminates, the program should nullify any reference from the old objects (lower) to the new objects (upper). As a result, the new objects become garbage to be collected. Often, however, old objects unexpectedly obtain additional references to new objects (Fig. 4).

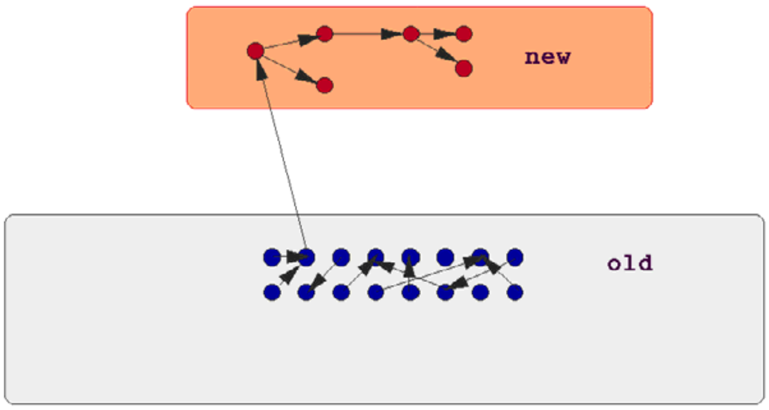


Fig. 3. Ideally, the program will remove any reference from old objects to new ones

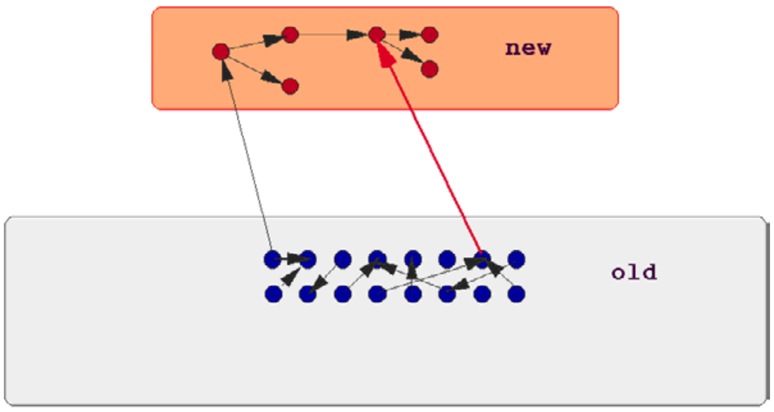
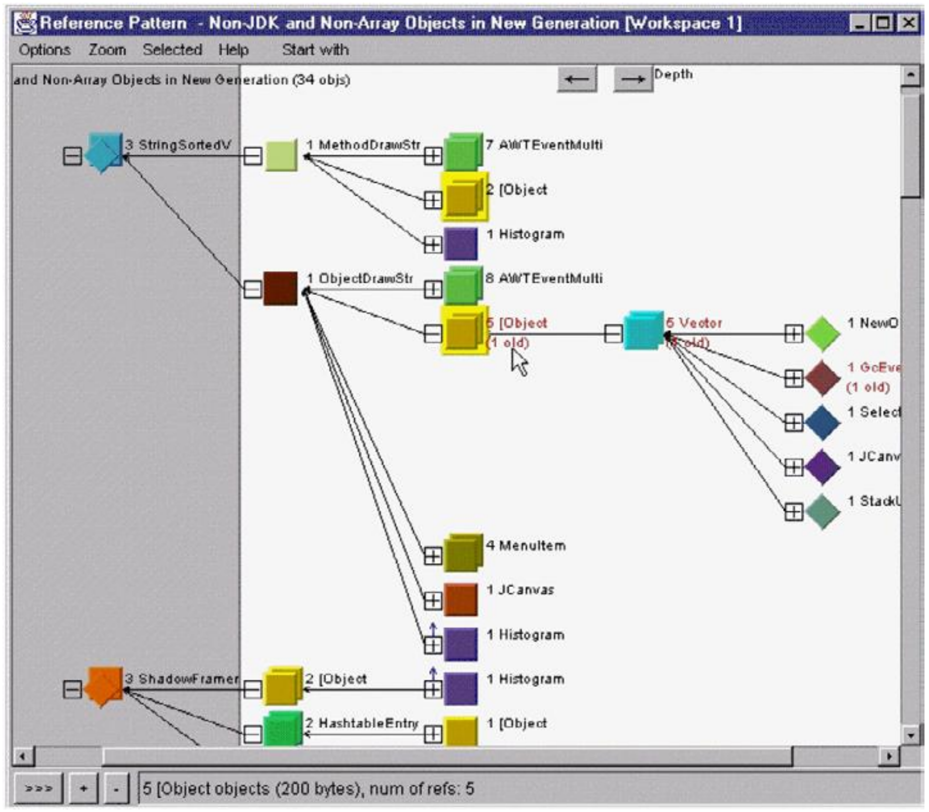


Fig. 4. Typically, old objects acquire unaccounted references

A typical case is a registry that acquires a reference to a new object (as shown by the long arrow on the right in Fig. 4). The programmer may not be aware of this reference and hence may fail to set this reference to null at the end of the operation. Thus the garbage collector will not reclaim the new objects.

To debug such a scenario with Jinsight, the programmer identifies a span of execution that subsumes the lifetime of the temporary objects (for example, the beginning and end of a user operation). The goal is to identify temporary objects, and any references to them, that persist beyond this period.

The Reference Pattern view (Fig. 5) helps you identify such objects; they appear in the gray area of the view. These are the temporary objects that are no longer needed



5 [Object objects (200 bytes), num of refs: 5

Fig. 5. Reference Pattern view again

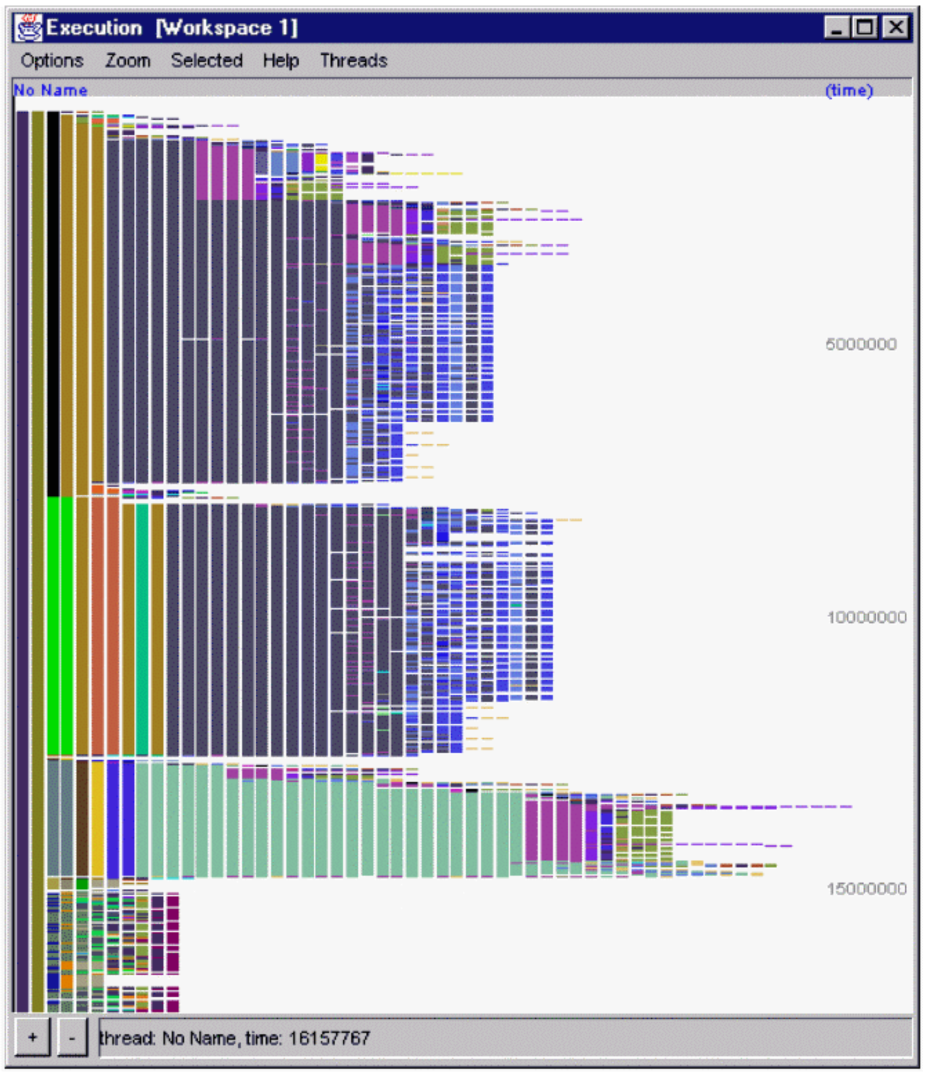
but cannot be reclaimed due to outstanding references. The white area on the right contains objects that refer, either directly or indirectly, to the unreclaimed objects. There are a few common sources of outstanding references to look for:

- References from old-generation objects (sometimes indirectly through other new-generation objects). Look for names labeled “old” in the white area, which identify old-generation objects.
- References from static data members. Look for diamonds in the white area, which represent class objects.

5 Performance Analysis by Visualizing Event Sequences

So far, we have focused on object profiling and memory leaks. Making a program run faster also requires understanding the structure and sequence of operations in the program. Performance analysis used to mean finding the hottest method in the program, but in most programs that approach would merely return one of Java’s String methods. Knowing the *sequence* of calls in a program is far more helpful.

Consider a sample program from the SQLJ package that retrieves data from a DB2 database. The program executes more slowly than desired. A better understanding of this program's execution can be had by examining it in Jinsight's Execution view (Fig. 6).



thread: No Name, time: 16157767

Fig. 6. Execution view

In this view, time proceeds from top to bottom. Each colored stripe represents the execution of a method on an object. Moving from left to right takes you deeper into the execution stack. The stripes are color-coded by class. A set (or "lane") of such stripes is displayed for each thread in a program. Lanes are arranged left-to-right. (Only one lane appears in Fig. 6.)

The length of each stripe (reflecting the time spent in each method), the number of stripes (corresponding to the stack depth), and the colors (denoting classes) characterize the behavior of each thread. We can also see the relative timing of events across threads by comparing lanes.

Four distinct phases are visible in Fig. 6. Passing the mouse cursor over each phase, the status line at the bottom of the view reveals the methods being invoked:

1. Loading the driver (*Class.forName*).
2. Making the connection to the database (*DriverManager.getConnection*).
3. Getting the primary key (*sample02_SJProfileKeys.getKey*).
4. Getting the results from the database.

Zooming into the final phase reveals a sequence of *next* and *println* method invocations, as shown in Fig. 7.

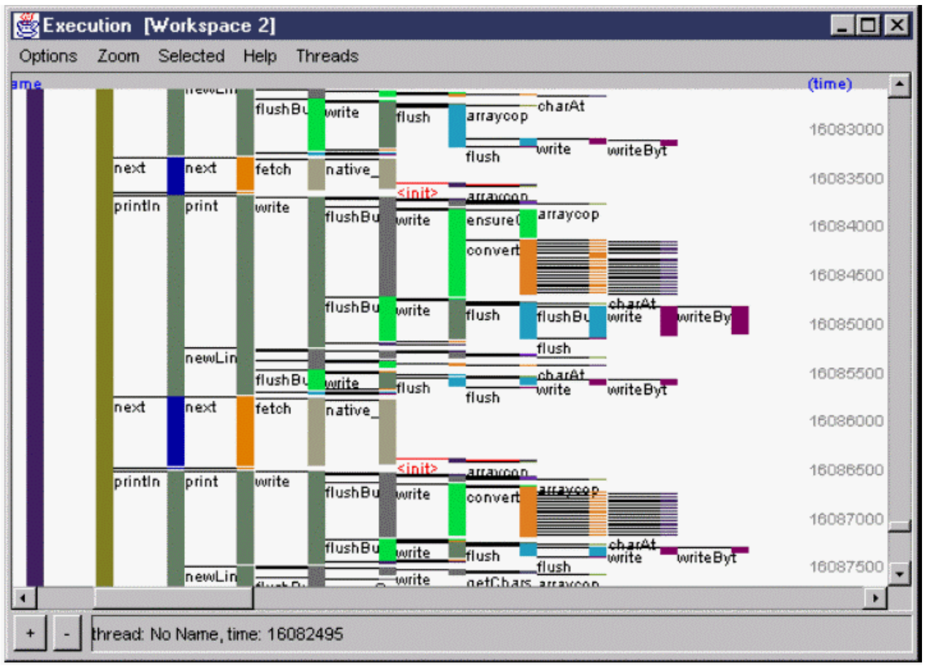
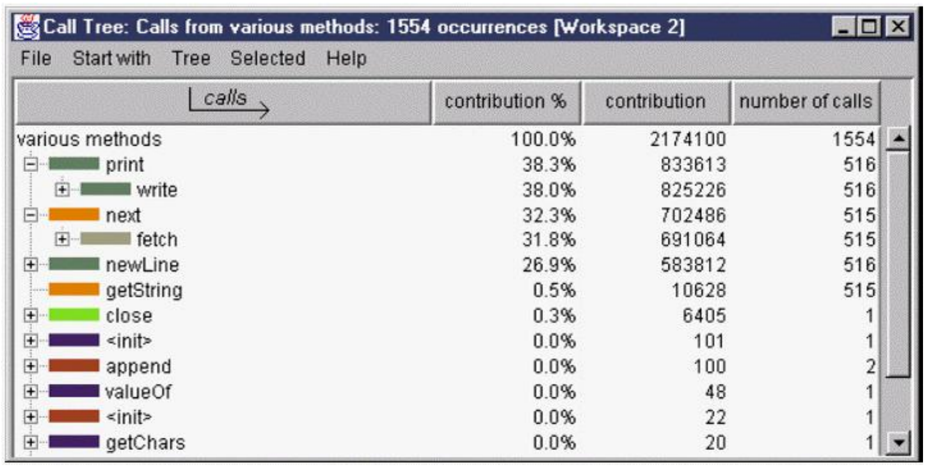


Fig. 7. Portion of final execution phase

Note that the *println* invocations (second stripe from left) are significantly taller than the intervening *next* invocations. This suggests that retrieving results from the database takes relatively little time. Substantially more time is spent printing them.

The Call Tree view can give us more precise numerical measurements. Fig. 8 shows that in the final phase of the program, 38% of the time is spent printing results, and 26% is spent printing newlines. Only 32% goes to retrieving the results.


Call Tree: Calls from various methods: 1554 occurrences [Workspace 2]
File Start with Tree Selected Help

calls	contribution %	contribution	number of calls
various methods	100.0%	2174100	1554
🟩 print	38.3%	833613	516
🟩 write	38.0%	825226	516
🟧 next	32.3%	702486	515
▬ fetch	31.8%	691064	515
🟩 newLine	26.9%	583812	516
🟧 getString	0.5%	10628	515
🟩 close	0.3%	6405	1
🟪 <init>	0.0%	101	1
🟫 append	0.0%	100	2
🟪 valueOf	0.0%	48	1
🟫 <init>	0.0%	22	1
🟪 getChars	0.0%	20	1

Fig. 8. Call Tree view

This analysis leads to two conclusions:

1. Initialization takes a lot of time (loading the driver, making the connection, etc.). It is beneficial therefore to pool database connections.
2. The retrieval times were much shorter than expected, thanks to being skewed by *print* statements.

6 Information Exploration Techniques

A common problem when analyzing program behavior is that information of interest is scattered, or mingled with unimportant information, making it hard to discern. The problem can show up as skewed numerical summaries or as visual overload in graphical views. In any case, key information may be hidden without the right organization.

Fortunately, the user often has valuable knowledge that can help. Jinsight lets users apply their knowledge to structure the information for visualization and measurement purposes [2]. Jinsight's capabilities in this vein are loosely inspired by techniques commonly found in tools such as OLAP systems for exploring complex databases in other application domains.

The user can select just a few items of interest (for example, certain unusual-looking invocations of a method) and navigate to other views to study the detailed or aggregate behavior of a particular activity. Users may also define their own analysis units, known as *execution slices*, to group related activities together, or to exclude activity that is outside the scope of study. Execution slices may be used in subsequent visualizations, for example, as a basis for measurement against other summary units such as threads, methods, individual invocations, and so forth There are many ways to define execution slices, ranging from a simple point-and-click in a given view to a full query capability based on static and dynamic attributes of trace data.

7 Tracing

Jinsight's visualizations are based on execution traces. To collect a trace, the user runs the target program with either a specially instrumented Java virtual machine (for pre-Java2 platforms) or with a profiling agent and a standard JVM (for Java2 platforms). The generated trace contains the details of the program's execution for a user-specified period of time. Jinsight users have successfully diagnosed numerous problems on large commercial applications with this approach.

Nevertheless, we have also encountered situations where this approach fell short. For example, when analyzing high-volume, Web-based applications in vivo, traces were often too large to visualize feasibly, and the overhead of generating the traces made the application deviate from its normal behavior. It was necessary to collect trace information more selectively.

One approach summarizes activity using aggregate statistics: histograms of heap consumption, tables of method invocation counts and average duration, and aggregate call graphs. But we found that statistics do not reveal enough of a program's dynamic structure (such as the sequence and context of method invocations) to support analyses for which Jinsight had already proven useful. Another approach is to filter broadly, limiting the trace to invocations of a particular method or class. However, broad filtering does not scale well, because the filtering criteria are not context-sensitive. For example, we are likely to be interested not in every String instantiation but only those driven by a particular type of transaction.

Jinsight supports *task-oriented tracing*, which can trace details of a program task selectively. Task-orientation admits only relevant details while retaining important contextual and sequencing information. Consider for example a high-volume transaction processing system. While analyzing database activity associated with a certain type of transaction, you can limit tracing to a few exemplary transactions and the activity they caused.

8 Related Work

Many systems present program execution information graphically. The work of Zeller, et al. [10, 11], is representative, focusing on the visualization of data structures and memory. Although the Reference Pattern view presents much the same information, Jinsight's focus is on the combination of control flow and data.

Zeller, et al., also support generalized graph layout for data structure visualization, within its inherent limitations. Others restrict their visualizations to standardized visual syntaxes. Mehner [12], for example, employs UML interaction diagrams to reveal concurrency problems in Java programs. The main limitation of such approaches lies in their scalability. UML and its competitors were designed in part to be easy for humans to draw, an irrelevant and limiting constraint in the context of execution visualization.

Still other visualization systems emphasize flexibility. BLOOM [8] has facilities for static and dynamic data collection and a wide range of data analyses, plus a visual query language for specifying the data to explore using a variety of 2D and 3D visualizations. Its scripting capabilities promote experimentation with new visualizations. Stasko, et al. [9], offer similar flexibility for experimenting with visualizations.

Systems like these are well suited to experimentation. In practice, however, it is more important to have a small number of visualizations optimized for particular tasks. For example, Jinsight helps reveal memory leaks through a simple process applied through a specialized visualization, the Reference Pattern view. Admittedly, it is impractical to expect a predefined view for every conceivable problem. But it is equally impractical to expect end-users to create effective views on their own, no matter how flexible the visualization environment. The key to this quandary is to foster synergies among predefined views, as Jinsight's navigation and slicing techniques afford. A small number of visualizations may thus address a combinatorial number of problems.

All this is predicated on visual abstractions that can deal with the glut of execution information. Various organizing abstractions have been used to filter execution information or group it into larger units. Sefika, et al. [13], use large architectural units, and Walker, et al. [14], introduce additional structural units as organizing principles. Dynamic relationships too are frequently used to organize information, for example, into call trees as in OptimizeIt [15]. Several systems also use queries of execution information to let the user filter out extraneous information and focus on an aspect of interest. The Desert system [16] provides a powerful query capability against static and dynamic information for various types of program understanding applications, as Hy+/GraphLog [17] does for analyzing distributed and parallel programs. Snodgrass [18] allows queries to be used for analyzing operating system behavior, and Lencenvicius [19] uses queries to debug live programs.

Our approach differs significantly from pure query-based systems. While powerful, those systems depend on their user to be sophisticated enough to set up meaningful visualizations and summary computations. By providing a query capability in the context of higher-level, domain-specific analysis units, Jinsight can introduce task-oriented specification, visualization, and summarization techniques that hide query language complexity from the user. Execution slicing coupled with view navigation offers proven scalability as well, having been employed successfully on production systems comprising over 5000 classes.

9 Future Work

Jinsight currently assumes the visualized application runs on a single JVM. Yet large production systems typically employ multiple JVMs, often across a network. Tracing and visualizing distributed Java programs presents new challenges: events have to be collected from different machines, timestamps might not be uniform, the system may have partial failures and concurrency problems, and many more. We are currently building a distributed version of Jinsight that will permit visualization across JVMs.

Real distributed systems are also heterogeneous, with middleware such as databases, HTTP servers, and message queuing systems in addition to Java components. We plan to extend our visualization environment to collect trace information from these non-Java sources. Our intent is to provide integrated visualizations for end-to-end transaction processing.

Creating Jinsight in an industrial research setting has given us the freedom to explore while having access to real customers who can put our prototypes through their paces. Indeed, fieldwork has driven our research. It reveals the problems developers are struggling with, and it helps us identify future problems and opportunities. Early customer trials are also invaluable for validating features. Some features may prove inadequate or irrelevant, while those crucial to solving certain problems prove lacking.

We exploit these characteristics through a three-stage development cycle lasting up to twelve months. First we exercise each new prototype ourselves on a difficult customer case, tweaking the prototype as needed. Then we release the prototype on the IBM Intranet for our colleagues' use, carefully noting their feedback. Finally, we release a version for general consumption on www.alphaWorks.ibm.com. Underused features, however intriguing, tend to be weeded out through this process, thus avoiding feature creep and ensuring the remainder has proven worth.

10 Conclusion

Jinsight offers unique capabilities for managing the information overload typical of performance analysis. Through a combination of visualization, pattern extraction, interactive navigation, database techniques, and task-oriented tracing, vast amounts of execution information may be analyzed intensively, making it easier to understand, debug, and tune programs of realistic size and running time.

Acknowledgements. We thank our colleagues Olivier Gruber, Ravi Konuru, Harini Srinivasan, and Mark Wegman for their contributions to Jinsight's development and for many valuable discussions. We are also grateful to our interns and to our users for their help and feedback.

References

[1] De Pauw, W., Mitchell, N., Robillard, M., Sevitsky, G., Srinivasan, H. Drive-by analysis of running programs. *Proceedings for Workshop on Software Visualization, International Conference on Software Engineering*, Toronto, Ontario, May 2001.
[2] Sevitsky, G., De Pauw, W., Konuru, R. An information exploration tool for performance analysis of Java programs. *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Europe)*, Zürich, Switzerland, March 2001, pp. 85-101.
[3] De Pauw, W., Sevitsky, G. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience* (2000) 12:1431–1454.
[4] De Pauw, W., Sevitsky, G. Visualizing reference patterns for solving memory leaks in Java. ECOOP '99, Lisbon, Portugal, June 1999. *Lecture Notes in Computer Science Vol. 1628*, Springer Verlag, pp. 116–134.
[5] De Pauw, W., Lorenz, D., Vlissides, J., Wegman, M. Execution patterns in object-oriented visualization. *Proceedings of the Fourth Conference on Object-oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998, pp. 219–234.

[6] De Pauw, W., Kimelman, D., Vlissides, J. Modeling object-oriented program execution. ECOOP ’94, Bologna, Italy, July 1994. *Lecture Notes in Computer Science Vol. 821*, Springer Verlag, pp. 163–182.
[7] De Pauw, W., Helm, R., Kimelman, D., Vlissides, J. Visualizing the behavior of object-oriented Systems, *OOPSLA ’93 Conference Proceedings*, Washington, D.C., September 1993, pp. 326–337.
[8] Reiss, S. An overview of BLOOM. *Proceedings of Program Analysis for Software Tools and Engineering (PASTE ’01)*, Snowbird, Utah, June 2001, pp. 2–5.
[9] Jerding, D. and Stasko, J. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, July–September 1998, 4(3):257–271.
[10] Zimmermann, T. and Zeller, T. Visualizing memory graphs (this volume).
[11] Zeller, A., Lütkehaus, D. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, January 1996, 31(1):22–27.
[12] Mehner, K. JaVis: A UML-based visualization and debugging environment for concurrent Java programs (this volume).
[13] Sefika, M., Sane, A., Campbell, R. Architecture-oriented visualization. *OOPSLA ’96 Conference Proceedings*. San Jose, California, October 1996. Published as *SIGPLAN Notices*, 31(10):389–405.
[14] Walker, R. J., Murphy, G., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J. Visualizing dynamic software system information through high-level models. *OOPSLA ’98 Conference Proceedings*, Vancouver, British Columbia, October 1998. Published as *SIGPLAN Notices*, 33(10):271–283.
[15] OptimizeIt Web Site, http://www.optimizeit.com/oproductinfo.html.
[16] Reiss, S. Software Visualization in the desert environment. *Proceedings of Program Analysis for Software Tools and Engineering (PASTE ’98)*, Montreal, Quebec, June 1998. pp. 59–66.
[17] Consens, M., Hasan, M., Mendelzon, A. Visualizing and querying distributed event traces with Hy+. *Lecture Notes in Computer Science, Vol. 819*, Springer Verlag, 1994, pp. 123–141.
[18] Snodgrass, R. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, May 1988, 6(2):157–196.
[19] Lencenvicius R., Hoelzle, U., Singh, A. K. Dynamic query–based debugging. ECOOP ’99, June 1999, Lisbon, Portugal. *Lecture Notes in Computer Science, Vol. 1628*, Springer Verlag, pp. 135–160.

JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs

Katharina Mehner

Department of Mathematics and Computer Science
University of Paderborn
D-33095 Paderborn, Germany
http://www.upb.de/cs/mehner.html
mehner@upb.de

Abstract.

Debugging concurrent Java programs is a difficult task because of multiple control flows and inherent nondeterminism. It requires techniques not provided by traditional debuggers such as tracing, visualization, and automated error analysis. Therefore, we have developed the JaVis environment for visualizing and debugging concurrent Java programs. The information about a running program is collected by tracing. The Unified Modeling Language (UML) is used for the visualization of traces. Traces are automatically analyzed for deadlocks. The tracing is implemented using the Java Debug Interface (JDI) of the Java Platform Debugger Architecture. The visualization is integrated into the UML CASE tool Together.

Keywords: object-oriented concurrent programming, tracing, dynamic program visualization, deadlock detection, Java, UML

1 Introduction

The object-oriented programming language Java is increasingly gaining importance in academia and industry mainly due to its platform independence and its role in internet computing. A specific characteristic of Java is the incorporation of concurrency into the language itself instead of supporting it with libraries [2]. Java thereby provides easy access to concurrency.

Nevertheless, concurrency is still a key factor for increasing the complexity of programs due to its multiple flows of control, also called threads, and its inherent nondeterminism. This makes the entire software development life cycle more difficult than in sequential object oriented programming. Companies who provide concurrent programming libraries have observed that their customers make many errors in using the libraries and have difficulties in finding their errors because there is no adequate tool support for debugging concurrent systems [3]. Existing debuggers especially lack support for errors specific to concurrent programs.

Typical concurrency errors involve more than one control flow and develop over a certain time while the program is running. A promising approach for debugging is *tracing*, i.e., making a protocol of the interesting events of a running program, because it captures the behavior over time. There is also a lack of using graphical means for debugging. The advantage of a *graphical visualization* is that it can depict complex dependencies within more than one dimension. To deal with the manifold information collected at runtime *automated* support to detect errors is required.

As this kind of support is also missing for Java we have developed the JaVis environment for *visualizing* and *debugging concurrent Java programs*. This environment covers tracing and visualization of sequential and concurrent Java programs. In addition, deadlocks are automatically detected and analyzed. The tracing uses the Java Debug Interface (JDI), an interface of the Java Virtual Machine (JVM) which is defined in the Java Platform Debugger Architecture [5]. The JDI allows to trace remote and already running programs. Using the JDI does not require to modify the source code, i.e., the tracing is *non-invasive*. The visualization is generated *post-mortem*, i.e., after the tracing is finished. The visualization is based on *UML sequence* and *collaboration diagrams* [9] which have been extended to model thread synchronization and deadlocks. The visualization is integrated into the standard UML CASE Tool Together [12].

This chapter is organized as follows. In Section 2 we discuss the requirements for debugging of concurrent programs in more detail. In Section 3 we introduce the JaVis environment from a user perspective. We describe the implementation of JaVis in Section 4. In Section 5 we discuss related work. We conclude and give an outlook in Section 6.

2 Motivation

Errors specific to concurrent programs are safety and liveness errors [7] such as deadlock or dormancy. These errors are difficult to detect because they do not occur in every program execution due to the inherent nondeterminism. In general, they can not be found by source code analysis with formal methods as this problem is NP-hard for arbitrary programs [6]. Therefore, in practice extensive testing is carried out to find concurrency errors in the running program.

At runtime these errors are characterized according to at least two dimensions. One dimension contains the different threads involved in the error, their *interactions*, and *dependencies*. The other dimension is the behavior over time because these errors typically involve more than one statement in each thread involved. This second dimension is also referred to as the *execution history*.

The detection of errors and localization of their reasons is usually done with the programmers' intuition. Programmers start by looking for typical patterns of behavior in the execution and then continue by looking for patterns in the code which can generate the execution patterns. These activities can be supported by the way in which information about the execution is represented, and they can be automated to a certain extent. The support provided by existing debuggers for concurrency errors has shortcomings. *Traditional debuggers* support stepping between program states and allow to inspect a program state in detail based on a textual representation. The execution history is only given implicitly by the sequence of program states. Single states or the entire sequence can not be saved. This makes it difficult to inspect behavior over time for different threads.

Tracing however, can capture the behavior over time, and thus, the development of an error over time. The trace format is usually textual for reasons of efficiency. The Java Dynamic Analyzer JaDA [1] generates textual traces for concurrent Java programs tracing method calls and thread synchronization. There are several approaches to generate traces: using frameworks, instrumenting the source code or the runtime environment, or other means of interacting with the runtime environment. Instrumentation at source level means that code is added to generate the trace. This can slow down threads and change the way they interact. Instrumentation or modifications for using a framework requires access to the source code, recompilation, and restart of the program. Companies often object to change the source code in order to generate traces. Source code is not always accessible, e.g., when tracing a vendor supplied binary component or an already running program. An embedded system or a server may be impossible or too costly to shut down. This is also the disadvantage of JaDA which requires to use framework classes to trace threads. Therefore, a *non-invasive* approach for tracing Java is needed.

Tracing is also used during *testing* because for testing a concurrent program not only the results of a test run have to be saved but also the execution history. Debugging starts from traces with errors. Therefore, debugging tools either have to integrate testing facilities or provide an import interface for traces. Debugging also requires to handle a large number of traces and the possibility to compare them. Therefore, *post-mortem* visualization is needed.

Textual traces have problems to show the dependencies between threads. Text is a linear structure and can not depict dependencies according to more than one dimension explicitly. A *graphical visualization* can depict execution history and complex dependencies according to at least two dimensions simultaneously. It is not advisable to develop new visualizations for traces from scratch. Instead, we aim at a better integration of visualization into the software development life cycle from the language perspective. The standard for visual modeling of object-oriented systems is the *Unified Modeling Language*(UML) [9], a set of languages for describing structure and behavior of software systems at different levels of abstraction. UML provides *extension mechanisms* for adaption to specific problem domains.

Also visualizations have to be manually analyzed for errors. Therefore, *automated* error detection and analysis is required. The result of the automated analysis can also be presented by a visualization. It is important for visualizing traces and for visualizing analysis results that the representation can be mapped back to the source code.

To summarize, we have identified the following requirements for tool support for debugging concurrent programs: independent non-invasive tracing, import of traces, post-mortem visualization of traces, UML-based the visualization, and analysis of concurrency errors. These requirements are supported by the JaVis environment.

3 The JaVis Environment

In this section we present the JaVis environment from the user perspective. The typical scenario of tracing a program and detecting a deadlock is illustrated. All UML diagrams shown have been generated with the UML CASE Tool Together.

3.1 The Running Example

The example is a banking simulation with six classes (see Figure 1). Employees of a bank use a terminal to start concurrent transactions such as a transfer between accounts. Concurrency in Java is based on class Thread [2,7]. From a thread object a single flow of control originates. Threads of the same program can share other objects. In order to avoid inconsistencies the access on these objects must be synchronized.

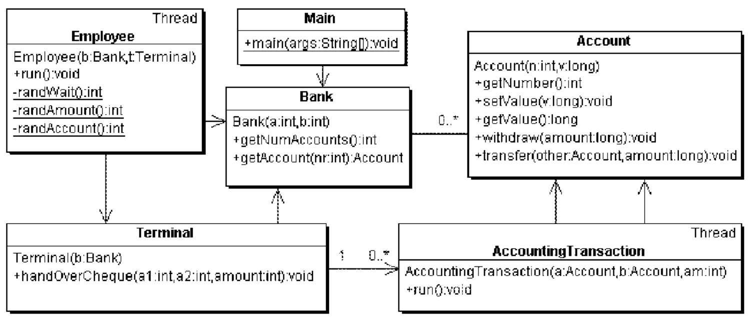


Fig. 1. UML class diagram

In the example, class Employee and AccountingTransaction inherit from the Java class Thread. Apart from these, class Main also has its own thread. All other classes are ordinary Java classes. In the following we will only focus on the classes AccountingTransaction and Account.

Class Account contains the balance of the account in value and the number in nr, initialized by the constructor. Method withdraw() withdraws money from the account and transfer() transfers money from another account to this account. Objects of class Account are used by threads and therefor have to be mutually exclusive. The key word synchronized is used for a method or a block within a method which needs exclusive access to the object on which it is called. Only non-synchronized methods or blocks can execute on the same object while a synchronized method or block is executed. To be able to enter a synchronized method or block a thread needs to obtain a lock on the object. If multiple threads are trying to obtain such a lock only one thread is assigned the lock. All other threads are blocked waiting for the lock. If a thread leaves a synchronized method or block, it releases the lock. By nondeterministic choice one of the blocked threads will receive the lock and can access the object. In the example, all methods of class Account are synchronized methods.

```
public class Account {
  private long value;
  private int nr;

  Account(int n, long v) {
        nr = n;
        value = v;
  }
  public synchronized int getNumber() {
        return nr;
  }
  public synchronized long getValue() {
        return value;
  }
  public synchronized void setValue(long v) {
        value = v;
  }
  public synchronized void withdraw(long amount) {
        value-=amount;
  }
  public synchronized void transfer(Account other, long amount) {
        other.withdraw(amount);
        value+=amount;
  }
}
```

A thread of class AccountingTransaction makes one transfer between two objects of class Account. When it is created the target and source account and the amount to be transferred are specified. A thread object is started by calling start() which internally calls the method run(). The run() method calls transfer() on the first account which subsequently calls withdraw() on the second account. When run() is completed the thread object is deleted.

```
public class AccountingTransaction extends Thread {
  Account a1;
  Account a2;
  int amount;

  AccountingTransaction(Account a, Account b, int am) {
        a1 = a;
        a2 = b;
        amount = am;
  }
  public void run() {
        sleep(10000);
        a1.transfer(a2,amount);
  )
}
```

We illustrate the behavior of run() with a sequence diagram (see Figure 2) generated by Together. Together can generate sequence diagrams from code for a single method but only for a sequential execution, i.e., the concurrent behavior of methods can not be illustrated.

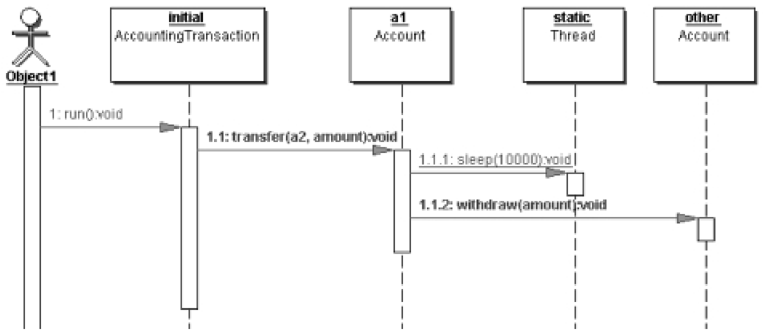


Fig. 2. Sequence diagram for method run() generated by Together

In the example, several threads of class AccountingTransaction transfer money between the same two accounts but in different directions. This scenario has the potential for a deadlock.

3.2 Tracing and Deadlock Detection

To test our example for deadlocks we run it with the JaVis Tracer from its beginning until a deadlock is detected. A small part of the textual trace is shown to demonstrate that it is difficult to read. In the JaVis Environment the user is not supposed to read the trace.

Each line of the trace contains a method entry, a method exit, or an attempt to enter a synchronized method where the object is already locked. The latter is called "Acquire" and needed for deadlock detection. The entries of each line are separated by ":". The first entry is the thread, identified by its class and its object ID. The second entry is the class and the ID of the object on which the method is called. In case of a method exit the callee is not identified which is denoted by a "-1". The next entry is the name of the method called. The following entry is "true" for a synchronized method and "false" otherwise. The last entry distinguishes method entry, exit or acquire. The host and the identification of the JVM which are also given for each thread are omitted here.

```
Employee@ID#90:Terminal@ID#-1:handOverCheque:false:Exit
AccountingTransaction@ID#97:AccountingTransaction@ID#97:run():false:Enter
AccountingTransaction@ID#97:Account@ID#79:transfer:true:Acquire
AccountingTransaction@ID#108:Account@ID#80:transfer:true:Acquire
AccountingTransaction@ID#108:Account@ID#80:transfer(Account(id=79), long 34):true:Enter
AccountingTransaction@ID#97:Account@ID#80:withdraw:true:Acquire
AccountingTransaction@ID#111:AccountingTransaction@ID#111:run():false:Enter
AccountingTransaction@ID#111:Account@ID#80:transfer:true:Acquire
AccountingTransaction@ID#108:Account@ID#79:withdraw:true:Acquire
```

After the last line displayed here a deadlock is detected and the program is halted. The thread with ID 97 of class AccountingTransaction acquires a lock which is locked by the thread with the ID 108. However, the thread with ID 108 acquires a lock for ID 79 which is locked by the thread with ID 97.

3.3 Trace Visualization

For the visualization of traces we chose UML [9] because it is well suited for visualizing program traces with *interaction diagrams*. These diagrams describe interaction, e.g., method calls, between objects over time on an instance level for multiple flows of control. While *sequence diagrams* have an explicit time axis *collaboration diagrams* render additional information about structural relationships and other dependencies. Apart from this, sequence and collaboration diagrams are equivalent. The timing order is presented by a hierarchical numbering of the messages. Here, it is preceded by a different number for each flow of control.

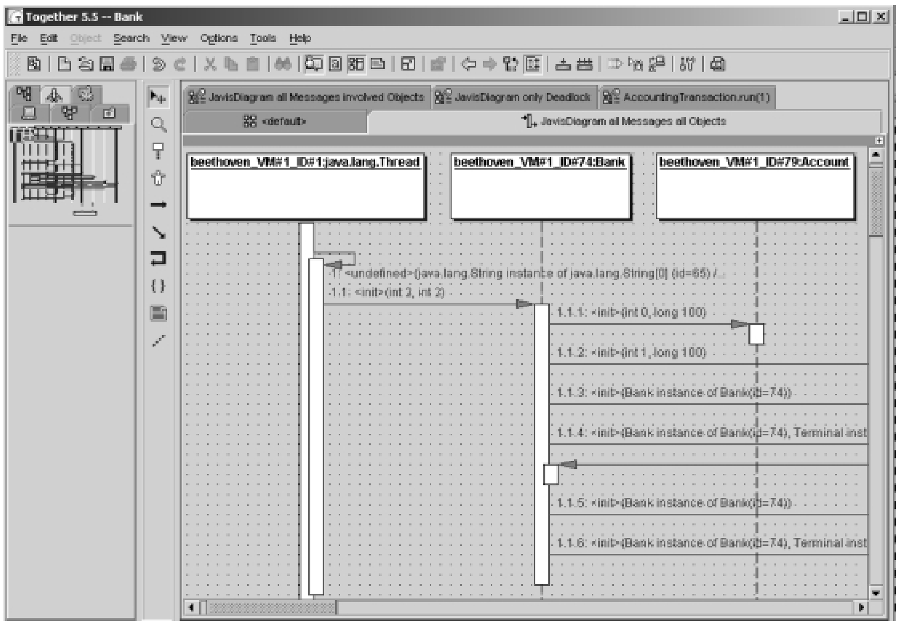

Together 5.5 -- Bank
File Edit Object Search View Options Tools Help
JavisDiagram all Messages involved Objects
JavisDiagram only Deadlock
AccountingTransaction.run(1)
<default>
JavisDiagram all Messages all Objects
beethoven_VM#1_ID#1:java.lang.Thread
beethoven_VM#1_ID#74:Bank
beethoven_VM#1_ID#79:Account
1: <undefined>(java.lang.String instance of java.lang.String[0] (id=65) / ...
1.1: <init>(int 2, int 2)
1.1.1: <init>(int 0, long 100)
1.1.2: <init>(int 1, long 100)
1.1.3: <init>(Bank instance of Bank(id=74))
1.1.4: <init>(Bank instance of Bank(id=74), Terminal inst
1.1.5: <init>(Bank instance of Bank(id=74))
1.1.6: <init>(Bank instance of Bank(id=74), Terminal inst

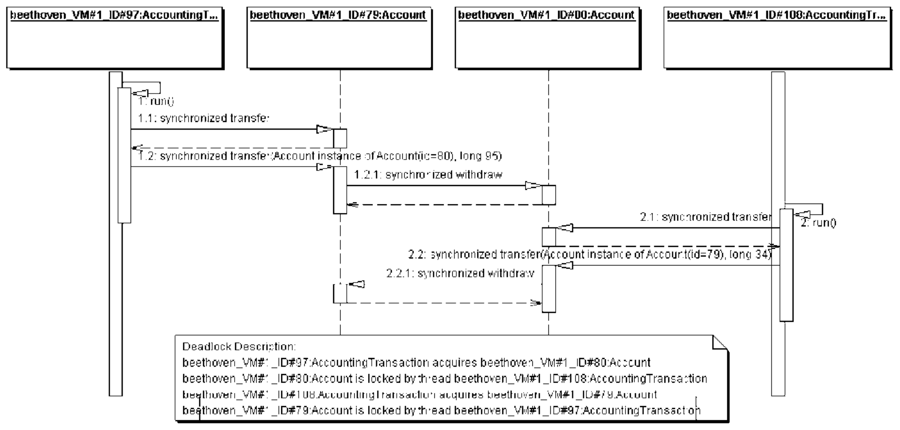
Fig. 3. Sequence diagram generated from trace

The first visualization generated from the trace is a sequence diagram of the entire execution (see Figure 3). The left column presents an overview. The very small black square is the selection displayed on the right side which shows the initialization phase with many constructors, displayed by init. Objects in a sequence or collaboration diagram have an identifier and a class. The identifier is generated from the trace by combining the host, the JVM and the internal object ID of the JVM. The class is given by the full path of the Java package. This sequence diagram contains somewhere the deadlock which is manually difficult to find. The diagram serves the purpose of an overview of the program execution over time. The equivalent collaboration diagram can be generated by Together but is too big to be useful.

3.4 Deadlock Visualization

The next step is to focus on the deadlock detected. The deadlock is visualized in an extra view with a sequence and a collaboration diagram showing only the objects directly involved in the deadlock.

In a *deadlock* at least two threads are blocked. Each thread locks an object and acquires the lock for the object already locked by the other thread. This situation can not be resolved by the threads involved. In Java deadlocks can occur when synchronized is used.

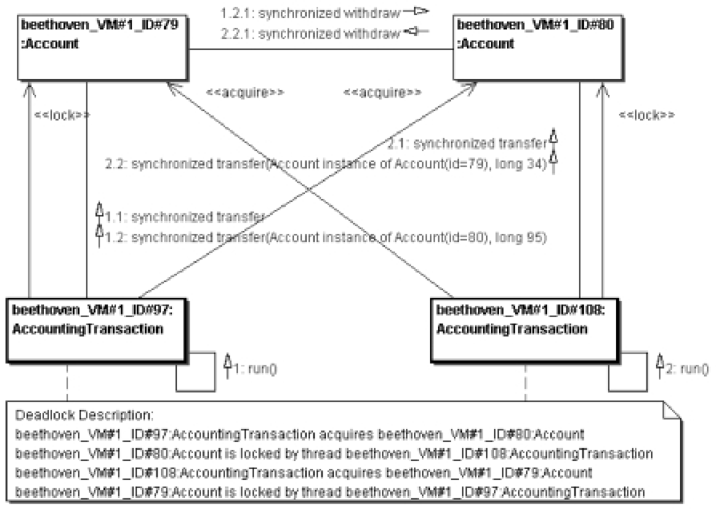

beethoven_VM#1_ID#97:AccountingT...
beethoven_VM#1_ID#79:Account
beethoven_VM#1_ID#80:Account
beethoven_VM#1_ID#108:AccountingTr...
1: run()
1.1: synchronized transfer
1.2: synchronized transfer(Account instance of Account(id=80), long 95)
1.2.1: synchronized withdraw
2: run()
2.1: synchronized transfer
2.2: synchronized transfer(Account instance of Account(id=79), long 34)
2.2.1: synchronized withdraw

Deadlock Description:
beethoven_VM#1_ID#97:AccountingTransaction acquires beethoven_VM#1_ID#80:Account
beethoven_VM#1_ID#80:Account is locked by thread beethoven_VM#1_ID#108:AccountingTransaction
beethoven_VM#1_ID#108:AccountingTransaction acquires beethoven_VM#1_ID#79:Account
beethoven_VM#1_ID#79:Account is locked by thread beethoven_VM#1_ID#97:AccountingTransaction

Fig. 4. Deadlock in sequence diagram

At first, we look at the sequence diagram for the deadlock (see Figure 4). When a synchronized method is called, first, the lock has to be obtained, and, after access is granted, the method body can be executed. These two steps are visualized. The AccountingTransaction with ID 97 calls transfer() on the Account with ID 79. The acquiring of the lock is visualized by the message 1.1:synchronized transfer() without parameters with an immediate return. Only when the lock is obtained the message 1.1:synchronized transfer() now with the actual parameters is drawn. If no second message with parameters is drawn the call is blocked. (This visualization was chosen because Together does not support shading of activation bars which would have been more intuitive to distinguish between the two steps of the message call.) When 1.2.1: synchronized withdraw() is called, this object is already locked by the other thread and so the thread is blocked an no second method call will be shown. A similar thing happens to thread AccountingTransaction with ID 108. The analysis of the deadlock is given in a rectangle containing a textual description of the deadlock, called a UML *note*. However, this diagram can not visualize the dependencies described in the analysis.

Now we focus on the corresponding collaboration diagram for the deadlock (see Figure 5). To show the dependencies involved in the blocking of threads the collaboration diagrams have to be extended to describe the operational meaning of synchronized: either the lock on an object is assigned or the thread is blocked waiting for the lock. In order to be able to describe Java locking relations at runtime we will use the UML extension mechanism of *stereotypes* to derive a new kind of modeling element. The stereotype name is rendered in guillemets and may have an associated graphical representation which can be used instead of the stereotype name.


1.2.1: synchronized withdraw
2.2.1: synchronized withdraw
beethoven_VM#1_ID#79 :Account
beethoven_VM#1_ID#80 :Account
<<lock>>
<<acquire>>
<<acquire>>
<<lock>>
2.1: synchronized transfer
2.2: synchronized transfer(Account instance of Account(id=79), long 34)
1.1: synchronized transfer
1.2: synchronized transfer(Account instance of Account(id=80), long 95)
beethoven_VM#1_ID#97: AccountingTransaction
beethoven_VM#1_ID#108: AccountingTransaction
1: run()
2: run()

Deadlock Description:
beethoven_VM#1_ID#97:AccountingTransaction acquires beethoven_VM#1_ID#80:Account
beethoven_VM#1_ID#80:Account is locked by thread beethoven_VM#1_ID#108:AccountingTransaction
beethoven_VM#1_ID#108:AccountingTransaction acquires beethoven_VM#1_ID#79:Account
beethoven_VM#1_ID#79:Account is locked by thread beethoven_VM#1_ID#97:AccountingTransaction

Fig. 5. Deadlock in collaboration diagram

We use stereotypes to define new kinds of dependencies between objects. When a synchronized method is called either the thread gets the lock or it is blocked, visualized by dependencies labeled "lock" and "acquire". When object 97:AccountingTransaction calls 1.2 synchronized transfer(), it locks it, which is visualized by the arrow from 97:AccountingTransaction to 79:Account labeled with "lock". When 1.2.1:synchronized withdraw() is called, this object is already locked by the other thread and so the thread is blocked, which is visualized by the arrow from 97:AccountingTransaction to 80:Account labeled "acquire". Using those arrows we can easily understand the deadlock by its cyclic dependencies. (In the tool those edges are red and green). We have introduced these extensions in [8].

4 Implementation of the JaVis Environment

The JaVis environment consists of two parts: a *tracing component* which allows to generate traces from a running Java program and to detect deadlocks, and a *visualization component* which generates several UML diagrams from the traces.

4.1 Tracing

The *Java Platform Debugger Architecture* (JDK Version 1.3) [5] provides the Java Debug Interface (JDI) implemented by each Java Virtual Machine (JVM). The JDI allows to collect debugging and tracing information from a running Java program without modifying the source code.

There are several possibilities to establish a connection between a tracing program, in the context of the Java Platform Debugger Architecture also called *debugger*, and the examined program, also called *debuggee*. The debugger can start the debuggee, either on the same or on a different JVM. The debugger can also be attached to an already running debuggee, either on the same or on a different JVM. These two options are used in our tracing component, implemented in Java based on the JDI. To collect runtime information the JDI offers methods to select events of interest and to provide callbacks which are executed when selected events occur. With the callbacks runtime information is delivered to the debugger. For our purpose we use the events for method entry and method exit. With these events information about the calling thread, the object called, and the parameters passed are delivered to the debugger. For each event it can also be decided to suspend the debuggee to get detailed information on the program state, e.g., the stack. We are also interested in the case when a thread is blocked on a synchronized method call. This is not an event in the JDI. Therefore, we use the suspension mechanism to find out which threads are blocked and in which method call they are blocked. However, this information is difficult to determine and only available on the Solaris JVM. From the information about method entry, method exit, and blocked method calls we generate a trace. The trace format was already described in Section 3.

At the moment, the deadlock detection is integrated in the tracing. The advantage is that a user not interested in the visualization can profit from the automated error detection. In future, it will also be available in the visualization component. For the deadlock detection during tracing we construct a graph describing the locking relations. When a thread acquires the lock for a synchronized object we insert the thread and the object as nodes and between them an *acquire*-edge. When a thread enters an synchronized object we insert an *lock*-edge and delete the acquire-edge if it existed. When a thread releases the lock for an object we delete the lock-edge. Whenever a thread attempts to enter an object already locked by another thread, we start a search for a cyclic dependency from the corresponding edge in the graph.

4.2 Visualization

We use the UML Case Tool Together [12] (Versions 4.01 and 5.5) to display the UML diagrams and their extensions. The support for concurrent visualizations is very little. Together can generate *sequential* sequence diagrams from the code of a Java method but not concurrent diagrams. The key word synchronized is not displayed and the notation for active objects which is useful for threads is not supported.

The Together Open API, a Java interface, allows to construct UML diagrams by instantiating classes for nodes and edges. Our visualization component first has to parse the trace. In the trace only the callee but not the caller is given for each event. For method exit not even the callee is given. However, this information is implicit in the trace. During parsing a runtime stack for each thread is simulated to determine caller and callee. Then the UML diagrams are generated via the Open API. One sequence diagram is generated for the entire trace. For each deadlock detected a sequence and a collaboration diagram are generated which contain only the objects and threads directly involved in the deadlock. The stereotypes "acquire" and "lock" have been visualized as pure text added to the message calls but not formally as new stereotypes.

We have implemented also a step by step display of the traces which can be navigated forwards and backwards, including jumping to selected statements. Temporary synchronization dependencies are displayed using "acquire" and "lock".

5 Related Work

Compaq Visual Threads [3] provides analysis of concurrency errors at runtime for POSIX threads, also called Pthreads, an IEEE standard supported by a library. Pthreads are not object-oriented, but Visual Threads can be used also from object-oriented languages including Java. The main goal is to assist thread programmers in detecting typical concurrency errors at runtime. Implemented are deadlock analysis, analysis for potential deadlocks based on monitoring the order of allocated resources, and many more including customizable analysis. Albeit its strength in the analysis the visualization of the results is only textual similar to the UML notes we were using.

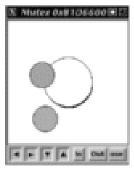


Fig. 6. Mutex View describing locking relations

The MutexView (see Figure 6) from the GThreads environment [13] visualizes the relation between threads and mutual exclusion locks. A big disk symbolizes a lock. The small disks symbolize the threads, distinguished by color. A thread having a lock is positioned inside the disk for the lock. Threads waiting for the lock are outside the disk for a lock. This view is animated but does not provide analysis of deadlocks. This view can show the access dependency graph only implicit by the positions of the disks. Therefore a deadlock is not really obvious. Another disadvantage of this visualization is the fact that the messages or calls which trigger the changes are not shown. This makes it difficult to relate a deadlock situation back to the code.

Jinsight [4,11] is a tool for tracing and visualizing Java programs based on the JDI. Concurrent Java programs are supported but not the analysis of concurrency errors. Jinsight uses diagrams similar to UML sequence diagrams which are also based on the object-oriented paradigm of sending messages between objects. In addition, information about the time consumed in different threads is given. The visualization can be used to manually look for errors.

A tool which uses UML is JAVAVIS [10]. It uses sequence diagrams and also object diagrams and includes threads. However, it can not display synchronization dependencies. The visualization is the interface for interactively executing the program step by step which works only for small programs. It is a tool for education and does not scale for real programs. Moreover, it does not provide filters or automated analysis for errors.

6 Conclusion and Outlook

In this chapter we have motivated the need for debugging support for object-oriented concurrent programs, especially for Java. We have presented the JaVis environment for visualizing and debugging concurrent Java programs based on UML.

Our main focus was to develop a deadlock detection and analysis based on tracing. Therefore, a trace format had to be defined which contains information about blocked threads. Then a technique had to be found which can detect these situation at runtime and can generate a corresponding trace. This was possible using the Java Debug Interface. It is a powerful means to collect information from the execution of Java programs. Even though the JDI does not directly support tracing of synchronization events, in all other respects it was easy to use. We hope that it will provide more features for concurrent debugging in future.

The other goal was to find a visualization based on a software engineering standard. Therefore, UML was used and extended. The general known problem that a language like UML does not scale well with a huge amount of information was not a concern for our purpose because we were applying means to reduce the amount of information by focussing on errors.

For the display of UML we also decided to chose a standard tool. Together provides already display facilities for UML diagrams but is limited in extension and presentation capabilities. This did hamper us a lot in implementing our visualization goals. Already the support for standard UML was a problem. Neither active objects where supported nor shading nor concurrent hierarchical numbering of messages. It was not possible to implement stereotypes in a reasonable manner. Therefore, we had to make many concessions. Doing it again, we would probably build our own visualization. However, we think that with little improvements the Together visualization can be used for empirical evaluation of the visual part of the JaVis environment.

Experiments with real systems have shown that the two JaVis components run very stable and scale to huge amount of data, also over the network. Especially the feature of tracing running remote systems was very helpful for selecting interesting parts of an application to make a trace.

This is work in progress. We want to address other liveness errors and safety errors. The idea is to widen the approach to address dynamic program comprehension of concurrent programs in general by providing more filters and views. With more functionality we are able to address evaluation. We want to compare our tool, a standard debugger, and programmer inserted printline statements because they are similar to tracing. First observations with students indicate that standard tracing is easier to interpret than manually inserted tracing output, especially for a peer programmer.

Acknowledgements. The author likes to thank Annika Wagner for support in the early stages of the project, Bernd Weymann for his major contribution to the implementation during his diploma thesis, and the unknown referees for their valuable comments.

References

1. A. Bechini and K. Tai. Design of a toolset for dynamic analysis of concurrent Java programs. In *Proc. International Workshop on Program Comprehension IWPC 1998*.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.
3. J. Harrow. Debugging multithreaded applications on Compaq Tru64 Unix operating systems. Technical report, Compaq Computer, 2000.
4. IBM. Jinsight. http://www.research.ibm.com/jinsight.
5. JavaSoft. Java Platform Debugger Architecture and Java Debug Interface. http://www.javasoft.com/products/jpda/.
6. E. Kraemer. Visualizing concurrent programs. In *Software Visualization: Programming as a Multimedia Experience*, pages 237–256. MIT Press, Cambridge, MA, 1998.
7. D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison Wesley, 1997.
8. K. Mehner and A. Wagner. Visualizing the synchronization of Java-threads with UML. In *IEEE Symposium of Visual Languages*, 2000.
9. Object Management Group. UML specification version 1.3, June 1999. http://www.omg.org.
10. Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic Program Visualization With Object and Sequence Diahrams Using The Jaba Debug Interface (JDI). In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
11. Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the Execution of Java Programs. in *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
12. TogetherSoft. Together. http://www.togethersoft.com.
13. Q. Zhao and J. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, Atlanta, GA, January 1995.

JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)

Rainer Oechsle and Thomas Schmitt

Institut für innovative Informatik-Anwendungen - I^{3}A
(Institute for Innovative Computer Applications)
FH Trier (University of Applied Sciences, Trier)
Postfach 1826
D-54208 Trier, Germany
{Oechsle, schmitto}@informatik.fh-trier.de

Abstract.

The goal of the JAVAVIS system is to help students understand what is happening in a Java program during execution. The primary focus of the first release is on sequential Java programs, although there is some support for visualizing concurrent threads. The system uses the Java Debug Interface (JDI), so there are no modifications needed in the Java source code for the extraction of information. The system shows the dynamic behavior of a running program by displaying several object diagrams and a single sequence diagram. There is one object diagram for each active method on the call stack. All modifications in the diagrams are done by smooth transitions.

1 Introduction

Education in Computer Science usually starts with an object-oriented programming language, and today this is typically Java. To our experience, however, programming novices have a hard time in learning, understanding, and mastering the different programming concepts, especially the object-oriented ones like class, object, method, and attribute.

In this paper we present a system called JAVAVIS that was developed as a tool to support teaching object-oriented programming concepts with Java, especially for the courses for beginners. The system can be used in the classroom by the teacher as well as by the students at home.

The tool monitors a running Java program and visualizes its behavior with two types of UML (Unified Modeling Language, [2]) diagrams which are de-facto standards for describing the dynamic aspects of a program, namely object and sequence diagrams. The users are able to step through the program line by line or method call by method call and watch the changes in the diagrams. All changes are done by smooth transitions. Otherwise it would be hard for the students to follow them.

In this paper we first describe JAVAVIS from a user's point of view in Section 2. The JAVAVIS system is based on two class libraries which are described in Section 3: First we give an overview of the Java Debug Interface (JDI) which is used by JAVAVIS for controlling a running Java program and for extracting information like e.g. the values of the local variables of a method. Second we present the Vivaldi Kernel, a Java class library for programming 2D animations with smooth transitions. Section 4 covers related work and Section 5 finally summarizes the paper and gives an outlook for further work.

2 JAVAVIS from a User's Perspective

In this section we show the execution and visualization of a simple program which adds several items to a singly-linked list and finally prints the list. The list items contain integer values.

2.1 Sample Program

The Java code of our sample program is as follows:

```
class ListItem
{
          int value;
          ListItem next;
}

public class List
{
   private ListItem first, last;

   public void append(int i)
   {
      if(first == null)
      {
         first = new ListItem();
         first.value = i;
         last = first;
      }
      else
      {
         last.next = new ListItem();
         last.next.value = i;
         last = last.next;
      }
   }

   public void print()
   {
      ListItem item = first;
      while(item != null)
      {
         System.out.print(item.value + " ");
         item = item.next;
      }
      System.out.println();
   }
   public static void main(String[] args)
   {
      List l = new List();
      for(int i = 0; i < args.length; i++)
      {
         int value = Integer.parseInt(args[i]);
         l.append(value);
      }
      l.print();
   }
}
```

The sample program consists of a class ListItem that has only two attributes, an integer value and a reference to the next item in the list. These attributes have package visibility and are thus accessible by methods of the following class List. Because we did not define explicitly any constructors, there is the standard constructor with no arguments for the class ListItem.

The class List has attributes which reference the first and last item of the singly-linked list of integer values. This class has also no explicitly defined constructor. So there exists a default constructor with no arguments for the List class as well. In addition, there are methods for appending a new list item at the end of the list and for printing the whole list in a single line. The append method distinguishes the two cases that the list is still empty (first == null) or not. The main method transforms each command line argument string into an integer and appends the value to the list which has been created before. Finally, the whole list is printed.

2.2 Executing and Visualizing the Sample Program

First, the program must be compiled with the option "–g" which means that debugging information is included in the class files.

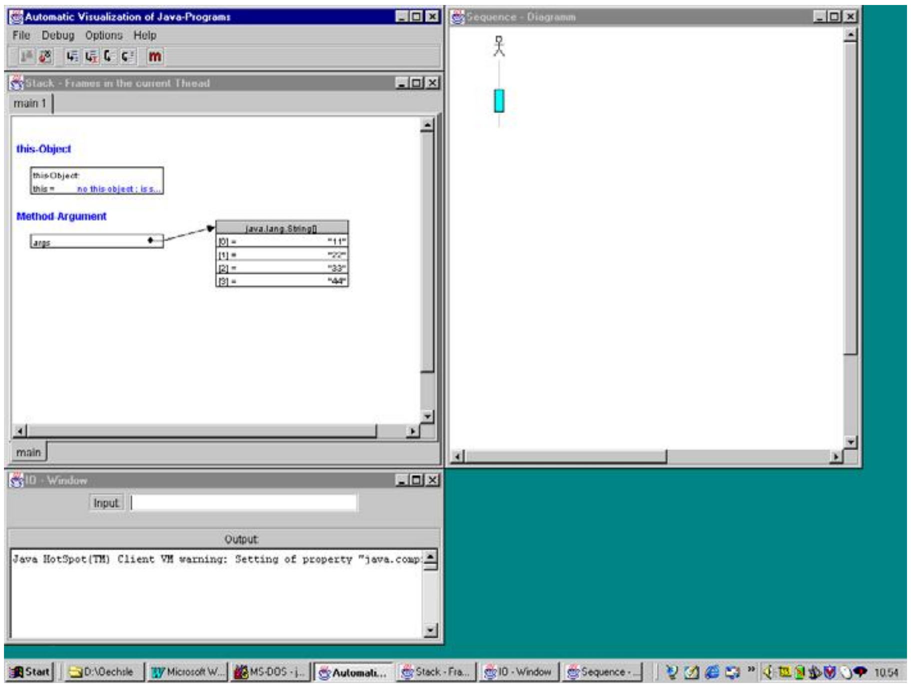
The visualization of a Java program can then be started in the same way as the execution of any Java program, except that we use the command `javavis` instead of `java`. So our sample list program may be started like this:

```
javavis List 11 22 33 44
```

After the start of the program four new windows appear on the screen, as shown in Fig. 1.

- The first window on the top left is for controlling the execution of the program. It contains menus and buttons in a toolbar for starting, stopping and restarting a program with possibly different command line arguments. In addition, there are commands for executing the next step of the running program. A next step may be a "step into a method", a "step over a method", a "step out of the currently executed method", and so on. There are also menus for setting the speed of the animations running in the object and sequence diagram windows, and for selecting the classes to be visualized.

- The window below the menu and toolbar window contains the object diagrams. We have an object diagram for each active method on the call stacks of the threads. At the bottom of the window the tabs show all currently existing object diagrams for a single thread. This means that the tabs mirror the call stack of this thread. At the top of this window there are tabs for selecting a thread. So if a tab is selected at the top, the tabs at the bottom will change. In Fig. 1 we see the situation after having entered the main method of the program. Therefore, there is only one thread, and the call stack of this thread contains only one method, the main method. Each object diagram shows all primitive data and all objects that are available at the moment. The diagram is divided into three parts: the this-object, the arguments of the method, and finally the local variables. In Fig. 1 there is no this-object because the main method is static. There is a single argument of the main method called args which is an array of strings that contains the command line arguments. We do not have any local variable.



```
Java HotSpot(TM) Client VM warning: Setting of property "java.comp
```

Fig. 1. The four windows of JAVAVIS after having entered the main method

- The bottom left window is for the standard input and output of the running program. A text field is used for input, and a text area for output.
- The window on the right contains the sequence diagram. In contrast to the object diagram window which may contain many object diagrams, there is only a single sequence diagram in this window. In Fig. 1 we see the activity of the main method. The column of the main method is marked by a person symbol.

After each program step there is an update of the sequence and object diagrams. All these modifications are done by smooth transitions. Unfortunately, the static nature of a paper makes it impossible to portray adequately the dynamics of the JAVAVIS system. After the execution of two iterations of the for-loop in the main method, e.g., the sequence diagram has changed; it looks as shown in Fig. 2.

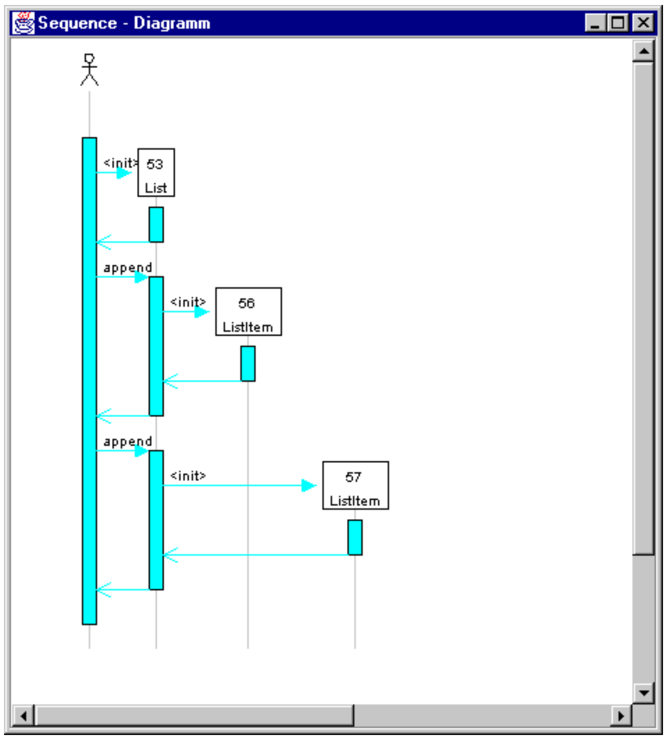


Fig. 2. Sequence diagram after execution of two iterations of the for-loop in the main method

We can clearly see in Fig. 2 that first a List object was created (arrow with <init> label) and the List constructor was executed (the default constructor). To this newly created List object the append method was applied. During the execution of the append method a new ListItem object was created and the ListItem constructor was called. The append method then was called a second time creating another ListItem object. The sequence diagram does not contain the calls of Integer.parseInt. This is because the system classes in packages like java.lang, java.util etc. are filtered out.

We now turn our attention to the object diagrams. If we start with the situation shown in Fig. 1 and execute the first step (i.e. the program enters the constructor of class List), the object diagram window is modified as shown in Fig. 3.

On the bottom of the window we recognize - by looking at the tabs - that a new object diagram has been pushed onto the previously existing diagram. The tab is labeled <init> which stands for the constructor. The object diagram represents the available data and objects of a method. In Fig. 3 we see the this-reference which points to the newly created list object to which the default constructor is applied. The graphical representation of the List object contains the attributes first and last which are both null.

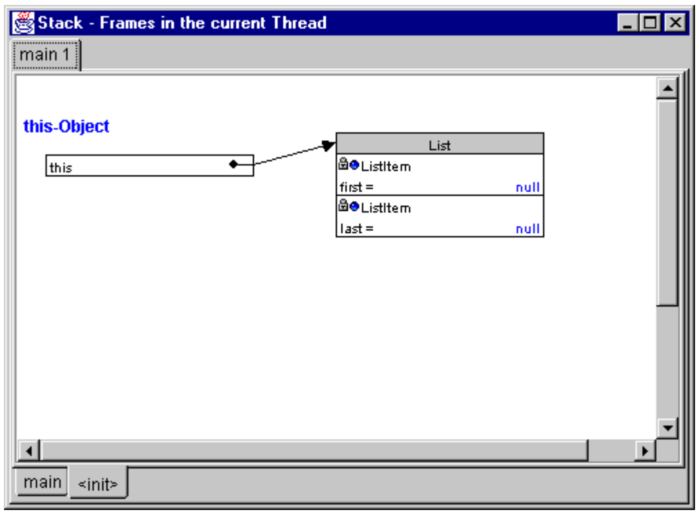


Fig. 3. Object diagram window after having entered the List constructor

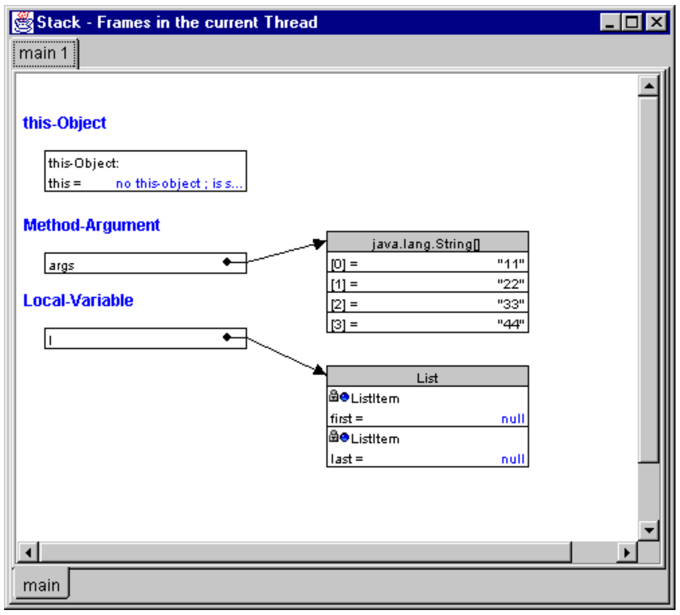


Fig. 4. Object diagram window after having assigned a reference of the newly created List object to the local variable l

After executing two more steps, namely returning from the constructor and assigning the reference of the new list object to the local variable l, the diagram object window looks as depicted in Fig. 4. It can be seen that the top level object diagram from Fig. 3 has disappeared, and that the object diagram for the main method now contains the local variable l and the newly created list object referenced by l.

On each execution of a program step, the sequence and object diagrams change accordingly. Fig. 5 presents a last object diagram snapshot for the execution of our sample program. The snapshot was taken between the call of Integer.parseInt and the call of the append method during the third iteration of the for-loop in the main method. The List object which is referenced by the local variable l now contains two ListItem objects with values 11 and 22. The local variable i holds the value 2, and the local variable value contains the parsed value of args[2], the third command line argument. In this moment the sequence diagram is as shown in Fig. 2. Execution of the next step, i.e. calling the append method, will push a new object diagram labeled append onto the object diagram stack. The this-reference will point to the same list as the local variable l in Fig. 5. The integer method argument i will have the value of 33. By further executing the program line by line, the students can follow all the steps taken for appending a new ListItem object containing 33 to the end of the list.

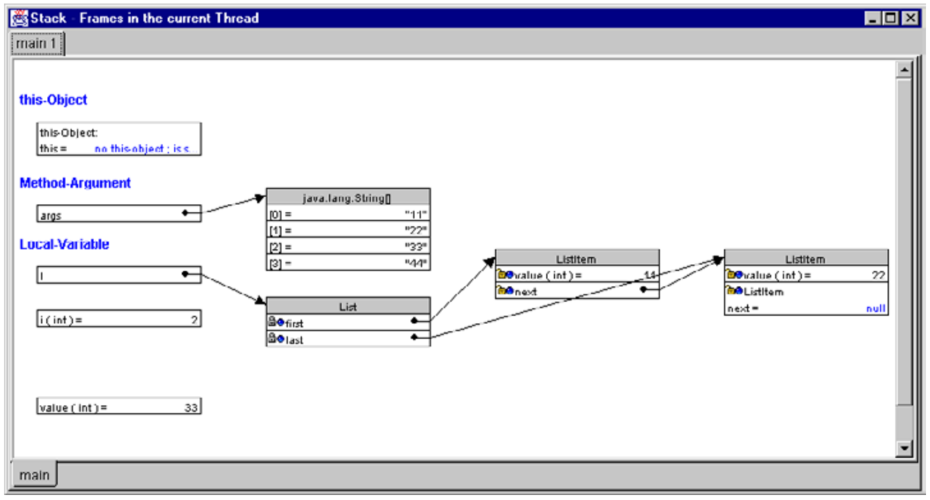


Fig. 5. Object diagram window during execution of the third iteration of the for-loop in the main method

2.3 Visualization of Concurrently Running Threads

Although the primary focus of the first release of JAVAVIS is on sequential Java programs, there is some support for visualizing concurrently running threads. As has been described already in Section 2.2, a stack of object diagrams is visible for each running thread, not only for the main thread. In addition, JAVAVIS is able to show the activity of more than one thread by coloring the activity boxes and the arrows in the sequence diagram differently for different threads. As a consequence, the concurrent usage of an object by several threads is visible in the sequence diagram.

In Fig. 6 an example for the visualization of the concurrent activity of two threads is presented. The main thread first creates a semaphore object. The class Other is a subclass of Thread with its own run method. After the main thread has created an object of type Other and started the thread, we can see an activity in the Other object. The execution of the run method applied to the Other object is drawn in a different (darker) color, because it is executed by a thread other than the main thread. The sequence diagram shows the classical synchronization of threads using a semaphore. The newly created thread calls the method p and is blocked. The main thread later calls v on the same semaphore, thus waking up the blocked thread. After the main thread has exited the v method, the other thread continues and exits the p method. The concurrent activity on the semaphore object by the two threads is easy to recognize.

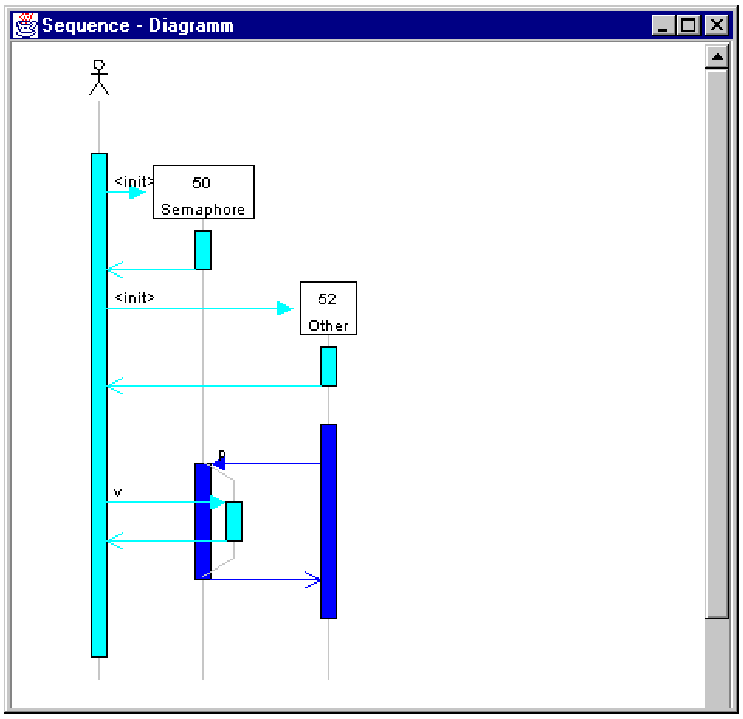


Fig. 6. Sequence diagram showing concurrently executing threads

The sequence diagram does not show blocked states of threads or locked states of objects. So, it is not possible to see that the semaphore object is locked on calling the synchronized p method, that this lock is released during the blocked state of a thread, that the semaphore object is locked again during the execution of the v method, and that the lock is reacquired before the blocked thread continues executing the p method. We are planning these kinds of visualization in further releases of JAVAVIS.

3 JAVAVIS from an Implementer's Perspective

JAVAVIS is based on two subsystems: the Java Debug Interface (JDI) for controlling and observing a running program, and the Vivaldi Kernel for representing the gathered information and building the animations with smooth transitions. In this section we first describe the overall architecture of JAVAVIS, and then we give a brief overview of both used subsystems.

3.1 Architecture of JAVAVIS

The overall architecture of the JAVAVIS is shown in Fig. 7.

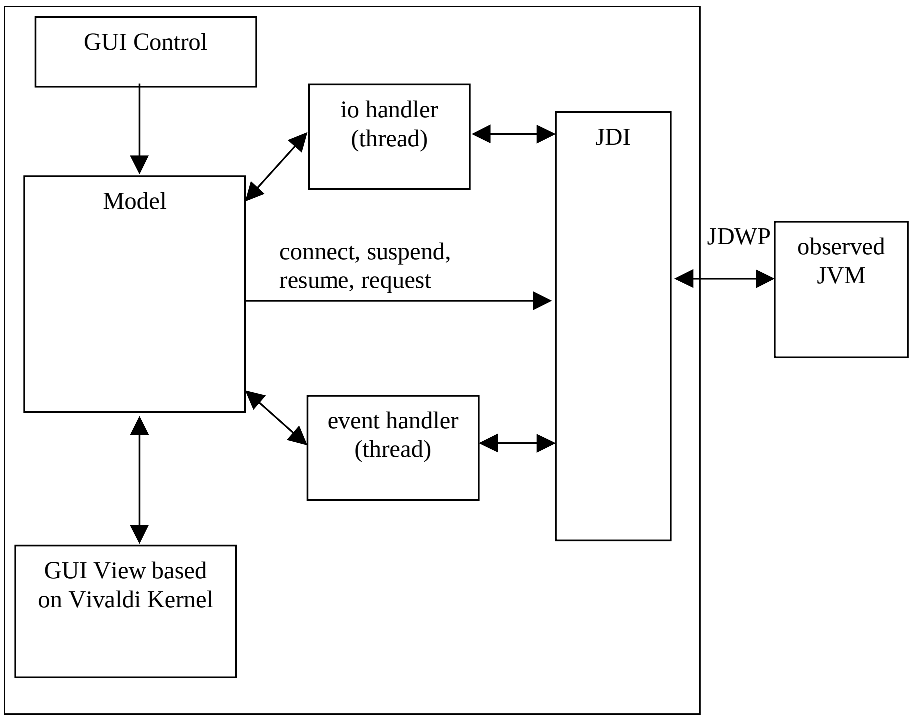


Fig. 7. Architecture of JAVAVIS

The system design follows the MVC (model – view – control) design pattern: there is a control component which contains the code for all the listeners that are called by the Java Swing components on user interaction (selection of a menu entry, clicking a button). The model component contains all data with the corresponding set and get methods. A change of the model data triggers an update of the view. The view is based on the Vivaldi Kernel class library for building animations with smooth transitions. The model component uses the JDI interface for sending commands to the observed program. Because reading standard output and removing events from the event queue are blocking operations, these activities are executed in separate threads. These threads inform the model component after an event has happened.

3.2 The Java Debug Interface

The Java Debug Interface is part of the Java Platform Debugger Architecture (JPDA). JDPA comprises three components (see Fig. 8):

- Java Virtual Machine Debug Interface (JVMDI): JVMDI is a native interface that each Java Virtual Machine (JVM) has to provide in order to be debugged.
- Java Debug Wire Protocol (JDWP): JDWP is a protocol for using the JVMDI from remote computers.
- Java Debug Interface (JDI): JDI is an interface implementation written in Java that accesses the JVMDI via JDWP. JDWP may be used over shared memory or over a TCP connection. The transport mechanism used by JDI can be chosen.

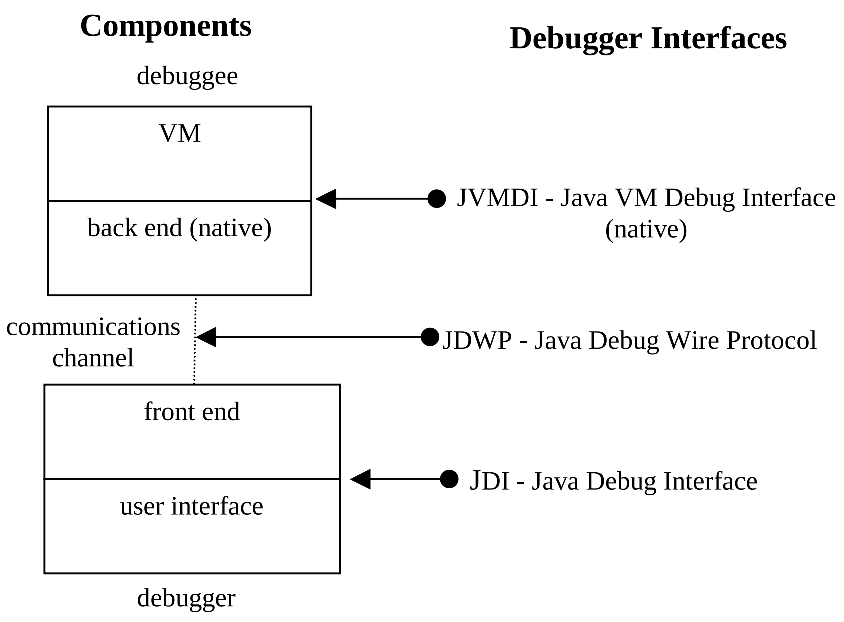


Fig. 8. The Java Platform Debugger Architecture (JPDA)

In the following we will call the side where the JDI is used the debugger, and the other side the debuggee. This means JAVAVIS plays the role of the debugger, and the observed program (e.g., the List program in the previous section) the role of the debuggee.

The connection between debugger and debuggee can be established in one of three ways:

- The debuggee is launched by the debugger using the JDI. The connection is automatically established between these two. This is the mechanism that is used by JAVAVIS.
- The debugger plays a client role and connects to an already running JVM.
- The debugger plays a server role and waits, until a JVM is opening a connection to it. In that case a JVM has to be started with a special option telling it to open a connection to a debugger.

The JDI allows to get an input and output stream. These streams can be used to read the standard output and write to the standard input of the debuggee. Moreover, the debugger can register its interest in certain events. When these events occur, the debuggee puts an event description into an event queue. The debugger can remove this event description from the event queue. It is blocked while the event queue is empty. The debuggee may be suspended after having sent an event information into the event queue. With additional commands, the debugger can resume the debuggee until the next event happens. While the debuggee is suspended, all kinds of state information can be read by the debugger.

3.3 The Vivaldi Kernel

The Vivaldi Kernel was developed within the project Vivaldi [8]. In this project we built several visualizations and animations for subjects of computer science education, e.g. synchronization and communication concepts like semaphores and message queues, the transport protocol TCP, the RIP routing protocol, and more. The Vivaldi Kernel forms the basis for all these visualizations. It consists of a class library for building animations with smooth transitions.

These classes are mainly classes for graphic elements and classes for attribute transitions.

Graphic Elements. All classes for the representation of graphic elements are subclasses of the class `GraphicElement`. These classes encapsulate the usual Java 2D graphic elements, but in a more object-oriented way. A color, e.g., is an attribute of a `GraphicElement` object, but in Java 2D a color is like a global variable that has to be set before drawing an element. There are some classes that have no direct peers within Java 2D, for example, the class `PhaseImageElement`. This class represents a sequence of JPEG or GIF images. There is an integer attribute called `phase` that specifies which image is displayed. Changing this attribute rapidly gives the impression of a movie for a suitable sequence of images. The class `GraphicElementGroup` forms a group of graphic elements. Because `GraphicElementGroup` is a subclass of `GraphicElement`, a group can contain other groups. So, the graphic elements form a tree. The design follows the well known design pattern "Composite" [6]. In Fig. 9 a part of the UML class diagram for the graphic element classes is depicted.

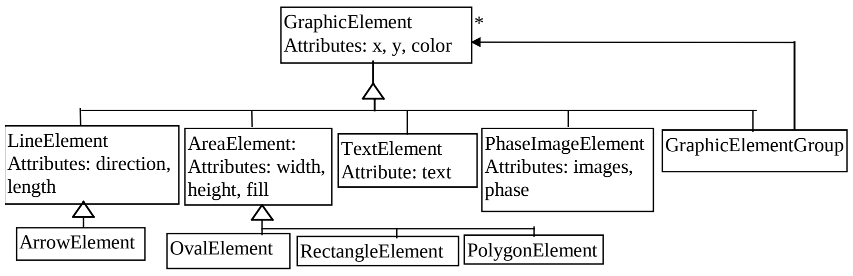


Fig. 9. Part of the class diagram for the graphic elements of the Vivaldi Kernel

A graphic element can react to mouse events (mouse clicked, mouse moved, etc.). According to the design pattern "Observer" [6] listener objects can be added to a graphic element. If an event occurs, the graphic element calls all registered listeners.

Attribute Transitions. The smooth transitions of the Vivaldi Kernel are based on rules for changing the attributes of the graphic elements. We call these rules attribute transitions. An attribute transition refers to a single attribute of a single graphic element. So in order to specify a movement of a graphic element, two attribute transitions are needed, one for changing the x and one for changing the y attribute. An attribute transition holds a reference to a graphic element and to a function which maps time to an attribute value. There are different transition classes for the different attributes of the graphic elements. In Fig. 10 an example is given for these relations. The upper part contains the tree of graphic elements. The lower part contains the attribute transitions. Each transition holds a reference to a graphic element and a time function. Transitions can also be grouped into transition groups. So the transitions also form a tree.

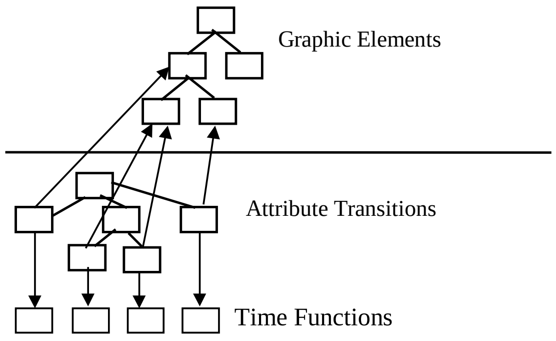


Fig. 10. An example for the relations between the tree of graphic elements, the tree of attribute transitions and the time functions

From Fig.10 we can see the following:

- A transition can reference a primitive graphic element or a graphic element group. It is thus easy to move a group of elements.

- There may be more than one transition that references a graphic element. This is useful if the transitions are of different types (i.e. modify different attributes of the graphic element like the x and y attribute, e.g.).
- Only primitive attribute transitions hold references to graphic elements and time functions. Attribute transition groups are only for grouping transitions (i.e. transition groups do not reference graphic elements or time functions).

Other facts about attribute transitions are:

- Transitions can be dynamically added to the tree of transitions and removed. It is thus possible to start and stop animations.
- Transitions can be finite or infinite. A finite transition is automatically removed from the tree of transitions after the transition has ended. This principle is recursively applied. A transition group ends if all members have ended.
- The transition class contains a method called `synch`. The method allows to wait for the end of a transition. If applied to a transition group, the calling thread is blocked until all transitions that are members of the transition group have ended.

4 Related Work

The work described in this paper belongs to the area of software visualization. Research in this area focuses on two subtopics: algorithm animation and program visualization. Algorithm animations are usually application-specific developments, where the abstractions are at a higher level ("problem oriented") and usually have to be done by human designers. Among the most popular algorithm animation systems are Balsa [3], Zeus [4], and Tango [9]. Most of our own developments within the project Vivaldi [8] can be classified as algorithm animations.

Program visualization systems, on the other hand, deal with lower level abstractions ("machine oriented"). The views of program and data structures can therefore be generated more or less automatically. These views, which do not require user intervention, are typically used for testing and debugging software. Program visualization systems can be classified as either visual debuggers [1], [7], [10], [11], [12] or visual program tracing systems. One of the most powerful visual program tracing systems is IBM's Jinsight [5] for Java programs that helps to tune the performance of a system or find memory leaks.

JAVAVIS is a program visualization system, as the title of this paper indicates. The most important differences between JAVAVIS and visual debuggers are as follows:

1. Although a debugger interface is used to extract the information that is visualized in JAVAVIS, our system is not a debugger. The goal of JAVAVIS is not to debug software, but help students to develop an understanding of object-oriented concepts.
2. Thus JAVAVIS is primarily suited for small-sized programs. All information that is accessible within a method is displayed. The students can see the "environment" of an executed method. Visual debuggers require the users to select the information that shall be visualized, because displaying all available information is not helpful in larger programs.

3. JAVAVIS provides at the same time a control flow view and a data view (sequence and object diagrams). Visual debuggers usually focus on visualization of data structures.
4. JAVAVIS consequently uses the UML diagram types and thus provides object-oriented views. Most visual debuggers are based on lower level abstractions.

JAVAVIS is different from a program tracing tool like Jinsight. A program tracing tool gathers lots of information from a running program and displays this information graphically in condensed forms. The experienced programmer may thus discover memory leaks or other effects. So it also helps to understand a program. JAVAVIS focuses on a much more detailed level. It is not a tool for experienced programmers, but for programming novices.

5 Summary and Outlook

In this paper we presented a system called JAVAVIS that was developed as a tool to support teaching object-oriented programming concepts with Java. The tool monitors a running Java program and visualizes its behavior with two types of UML diagrams which are de-facto standards for describing the dynamic aspects of a program, namely object and sequence diagrams. The users are able to step through the program line by line or method call by method call and watch the changes in the diagrams. All changes are done by smooth transitions.

JAVAVIS has been used successfully in lectures. We have not polled our students in a systematic way about their usage and their impression of JAVAVIS. However, we got very positive feedback from some of them. Our main goal, shaping their thinking about a running Java program, seems to work excellently.

We believe that it was a good decision to use UML diagrams in our program understanding tool. We are convinced that a programming language like Java and a modeling language like UML should be introduced and used together. Using UML not only in the design phase of a software engineering project, but also in the understanding, testing and debugging phase may be a good vehicle for students to better understand and get used to UML.

We also believe that it was a good decision to base the JAVAVIS system on JDI and the Vivaldi Kernel. All needed functions are provided by these systems. If we had to implement JAVAVIS again, it would be based on theses systems once more.

Still, some items remain to be done:

- Students would like to see the source code of the running program in a separate window, with the currently executed line highlighted. This is easy to do, but it has to be done.
- JAVAVIS does not scale for larger programs. The current version has a built-in filtering option. This option, however, is based on classes and is therefore too coarse-grained. We would like to have finer-grained techniques.
- After having acquired some knowledge about object-oriented programs, students will have a second hard time when threads are introduced in their computer science education. So we would like to have better support for visualizing concurrent threads. As already mentioned in Section 2.3, currently JAVAVIS

does not show blocked states of threads or locked states of objects. We would like to remedy this weakness in future releases. In addition, it would be nice if the system would show the calls of the wait, notify and notifyAll methods and their effects.

References

1. Baeza-Yates, R.A., Quezada, G., Valmadre, G.: Visual Debugging and Automatic Animation of C Programs. In: Eades, P. , Zhang, K. (eds.): Software Visualization, World Scientific Press (1997)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
3. Brown, M.H.: Exploring Algorithms Using Balsa-II. Computer, Vol. 21, No. 5 (May 1988) 14-36
4. Brown, M.H.: Zeus: A System for Algorithm Animation and MultiView Editing. Proceedings IEEE Workshop Visual Languages, IEEE CS Press, Los Alamitos, California (1991) 4-9
5. De Pauw, W., Mitchell, N., Robillard, M., Sevitsky, G., and Srinivasan, H.: Drive-by Analysis of Running Programs. Proceedings for Workshop on Software Visualization, International Conference on Software Engineering, Toronto (May 12-13, 2001)
6. Gamma, E., Helm, E., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Hanson, D.R., Korn, J.L.: A Simple and Extensible Graphical Debugger. In: Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, California, USA (January 1997)
8. Oechsle, R., Becker, R.: Das Projekt Vivaldi (VIsualisierung Von Ausgewählten Lehrinhalten Der Informatik), 15. GI/ITG-Fachtagung "Architektur von Rechensystemen" (ARCS '99) und "Arbeitsplatzrechensysteme" (APS '99), Friedrich-Schiller-Universität Jena, (October 1999) 263-270 *in German*
9. Stasko, J.T.: Tango: A Framework and System for Algorithm Animation. IEEE Computer, Vol. 23, No. 9 (September 1990) 27-39
10. Stasko, J.T., Mukherjea, S.: Towards Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger. ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3 (September 1994) 215-244
11. Zeller, A.: Visual Debugging with DDD. Dr. Dobb's Journal #322 (March 2001) 21-28
12. Zeller, A., Lütkehaus, D.: DDD – A Free Graphical Front End for UNIX Debuggers. ACM SIGPLAN Notices, Vol. 31, No. 1 (January 1996)

Visualizing Memory Graphs

Thomas Zimmermann[1] and Andreas Zeller[2]

[1] Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, 94032 Passau
zimmerth@fmi.uni-passau.de

[2] Universität des Saarlandes, FR Informatik
Lehrstuhl für Softwaretechnik
Postfach 15 11 50, 66041 Saarbrücken
zeller@cs.uni-sb.de

Abstract.

To understand the dynamics of a running program, it is often useful to examine its state at specific moments during its execution. We present *memory graphs* as a means to capture and explore program states. A memory graph gives a comprehensive view of all data structures of a program; data items are related by operations like dereferencing, indexing or member access. Although memory graphs are typically too large to be visualized as a whole, one can easily focus on specific aspects using well-known graph operations. For instance, a greatest common subgraph visualizes commonalities and differences between program states.

Keywords: program understanding, debugging aids, diagnostics, data types and structures, graphs

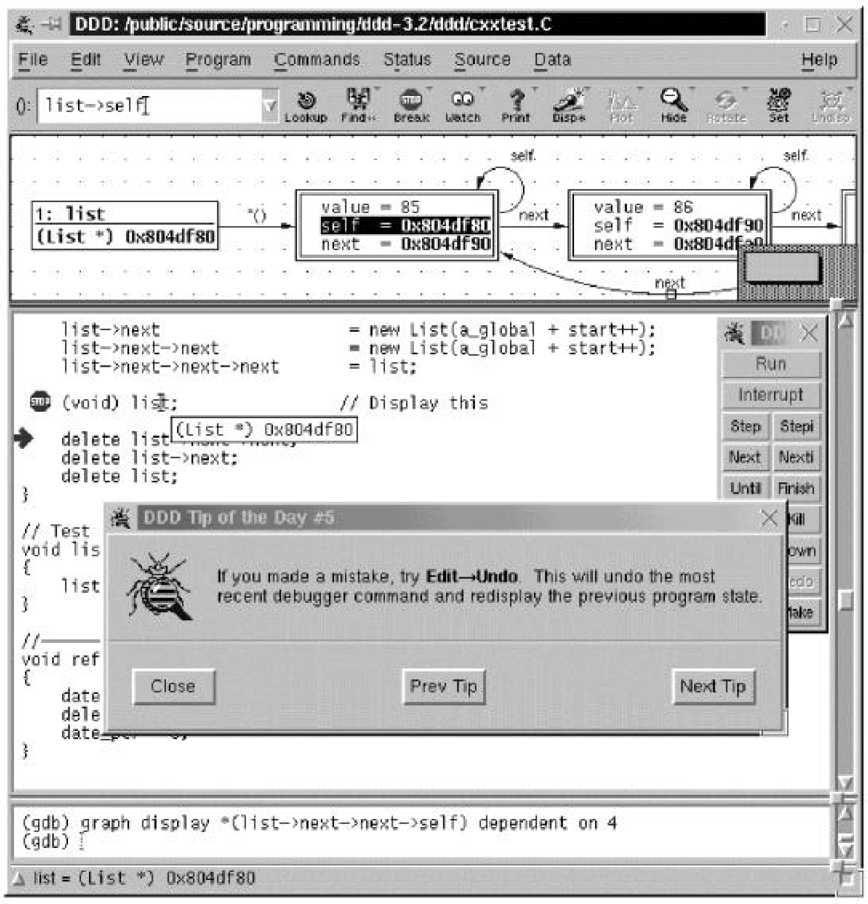
1 A Structured View of Memory

Exploring the state of a program, to view its variables, values, and current execution position, is a typical task in debugging programs. Today's interactive debuggers allow accessing the values of arbitrary variables and printing their values. Typically, values are shown as *texts*. Here's an example output from the GNU debugger GDB:

```
(gdb) print *tree
*tree = {value = 7, _name = 0x8049e88 "Ada", _left = 0x804d7d8,
  _right = 0x0, left_thread = false, right_thread = false,
  date = {day_of_week = Thu, day = 1, month = 1, year = 1970,
   _vptr. = 0x8049f78 ⟨Date virtual table⟩}, static shared = 4711}
(gdb) _
```

Fig. 1. Printing textual data with GDB

Although modern debuggers offer graphical user interfaces instead of GDB's command line, data values are still shown as text. This is useful in the most cases, but fails



```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;                // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date_
}
```

(List *) 0x804df80

DDD Tip of the Day #5

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Fig. 2. The GNU DDD debugger

badly as soon as references and pointers come into play. Consider Figure 1, for example: where does `tree->_left` point to? Of course, one can simply print the dereferenced value. However, a user will never notice if two pointers point to the same address—except by thoroughly checking and comparing pointer values.

An alternative to accessing memory in a name/value fashion is to model memory as a *graph*. Each value in memory becomes a vertex, and references between values (i.e. pointers) become edges between these vertices. This view was first explored in the GNU DDD debugger front-end [8], shown in Figure 2.

In DDD, displayed pointer values are dereferenced by a simple mouse click, allowing to unfold arbitrary data structures interactively. DDD automatically detects if multiple pointers pointed to the same address and adjusts its display accordingly. DDD has a major drawback, though: each and every pointer of a data structure must be dereferenced

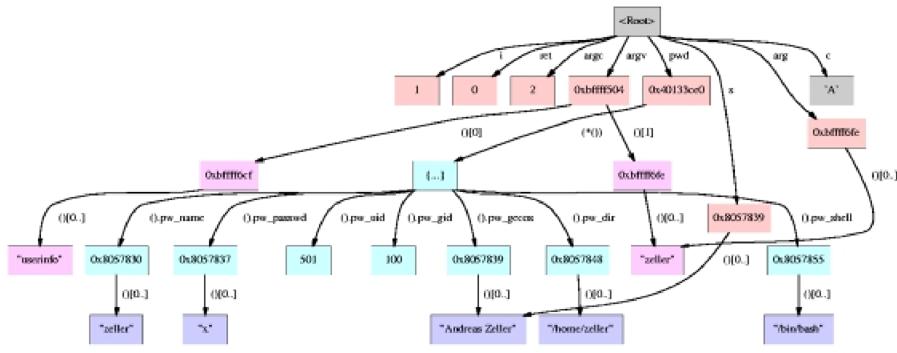


Fig. 3. A simple memory graph

manually. While this allows the programmer to set a focus on specific structures, it is tedious to access, say, the 100th element in a linked list.

To overcome these limitations, we propose *memory graphs* as a basis for accessing and visualizing memory contents. A memory graph captures the program state as a graph, very much like DDD does; however, it is extracted automatically from a program. Since a memory graph encompasses the entire program state, it can be used to answer questions like:

– Are there any pointers pointing to this address?
– How many elements does this data structure have?
– Is this allocated memory block reachable from within my module?
– Did this tree change during the last function call?

In this paper, we show how to extract and visualize memory graphs, sketching the capabilities of future debugging tools.

2 An Example Graph

As a simple example of a memory graph, consider Figure 3. The memory graph shows the state of a program named *userinfo*[1]. *userinfo* takes a UNIX user name as argument and shows the user's full name and e-mail address.

In our case, *userinfo* was invoked with *zeller* as argument. You can see this by following the edge named "*argv*" (the program's arguments). Dereferencing the first element of *argv* (following the edge labeled "()[1]", we find *argv*[1]—the argument "*zeller*".

To fetch the full name, *userinfo* accesses the user database via its *pwd* variable. By dereferencing the link named *pwd* from the top, you find a node named "{ . . . }". This is the record *pwd* points to (i.e. $*pwd$). Further descendants include the user id, the group id, the full name of the user and the UNIX user name "*zeller*" as well). You see that the

[1] *userinfo* is part of the GNU DDD distribution.

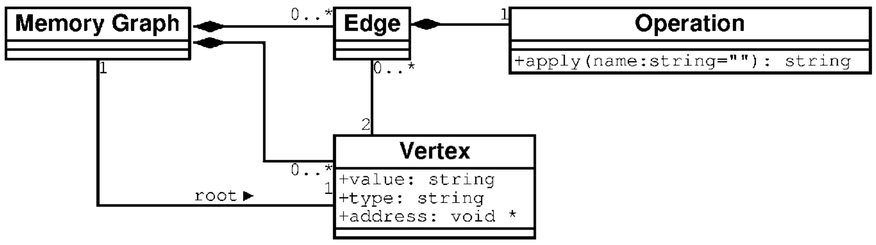


Fig. 4. Memory graphs UML object model

two strings *argv*[1] and *pwd* $\rightarrow$ *pw_name* are disjoint; however, *arg* points to the same string as *argv*[1]. Obviously, such a graph drawing is much more valuable than, say, a table of variables and values.

If you are interested in a formal definition of memory graphs, see Figures 4 and 5.

3 Obtaining Memory Graphs

How does one obtain a memory graph? Figure 6 gives a rough sketch. At the bottom is the debuggee, the program to be examined. Its state is accessed via a standard debugger such as GDB. The memory graph extractor queries GDB for variable names, types, sizes, and values. Since GDB is controlled via the command line, the dialogue between memory graph extractor and GDB is actually human-readable, as shown in Figure 7 (bold face stands for GDB commands as generated by the memory graph extractor).

You can see how the memory graph extractor queries GDB for the address and size of the *pwd* variable, then, having found it is a pointer, queries the object pointed to by dereferencing *pwd*. The object *pwd* points to is a C struct (a record), so the memory graph extractor goes on querying the addresses, sizes and values of the individual members. Note the usage of an internal GDB variable $\$v_{17}$ here; this is done to avoid the transmission of long expression names (such that we can use, say, $\$v_n$ $\rightarrow$ *value* instead of *list* $\rightarrow$ *next* $\rightarrow$ *next* $\rightarrow$ *next* $\rightarrow \cdots \rightarrow$ *value*)

Once the entire graph is extracted, it can be made available for the programmer to display or examine; it can also be shown in a debugging environment where additional manipulations become available.

The formal details of obtaining memory graphs are listed in Figure 8; special caveats about C programs are given in Figure 9.

4 Querying Memory Graphs

Once a memory graph is obtained, one can apply a number of graph algorithms to search for specific variables and paths. Useful operations include:

The Formal Structure of Memory Graphs

Let $G = (V, E, \mathit{root})$ be a memory graph containing a set V of vertices, a set E of edges, and a dedicated vertex *root* (Figure 4):

Vertices. Each vertex $v \in V$ has the form $v = (\mathit{val}, \mathit{tp}, \mathit{addr})$, standing for a value *val* of type *tp* at memory address *addr*.

As an example, the C declaration

```
int i = 42;
```

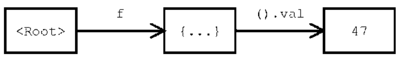
results in a vertex $v_{\texttt{i}} = (42, \text{int}, \mathit{0x1234})$, where *0x1234* is the (hypothetical) memory address of `i`.

Edges. Each edge $e \in E$ has the form $e = (v_1, v_2, \mathit{op})$, where $v_1, v_2 \in V$ are the related vertices. The operation *op* is used in constructing the expression of a vertex (see below).

As an example, the C declaration of the record ("struct") `f`,

```
struct foo { int val; } f = {47};
```

results in two vertices $v_f = (\{\ldots\}, \text{struct foo}, \mathit{0x5678})$ and $v_{f.val} = (47, \text{int}, \mathit{0x9abc})$, as well as an edge $e_{f.val} = (v_f, v_{f.val}, \mathit{op}_{f.val})$ from v_f to $v_{f.val}$:



Root. A memory graph contains a dedicated vertex $\mathit{root} \in V$ that references all base variables of the program. Each vertex in the memory graph is accessible from root.

In the previous examples, `i` and `f` are base variables; thus, the graph contains the edges $e_i = (\mathit{root}, v_i, \mathit{op}_i)$ and $e_f = (\mathit{root}, v_f, \mathit{op}_f)$.

Operations. *Edge operations* construct the name of descendants from their parent's name.

In an edge $e = (v_1, v_2, \mathit{op})$, each operation *op* is a function that takes the expression of v_1 to construct the expression of v_2. We denote functions by $\lambda x.B$—a function that has a formal parameter x and a body B. In our examples, B is simply a string containing x; applying the function returns B where x is replaced by the function argument.

Operations on edges leading from *root* to base variables initially set the name; so $\mathit{op}_i = \lambda x.\texttt{"i"}$ and $\mathit{op}_f = \lambda x.\texttt{"f"}$ hold.

Deeper vertices are constructed based on the name of their parents. For instance, $\mathit{op}_{f.val} = \lambda x.\texttt{"}x\texttt{.val"}$ holds, meaning that to access the name of the descendant, one must append `".val"` to the name of its parent.

In our graph visualizations, the operation body is shown as edge label, with the formal parameter replaced by `"()"` (that is, we use $\mathit{op}(\texttt{"()"})$ as label). This is reflected in the figure above.

Names. The following function *name* constructs a name for a vertex v using the operations on the path from v to the root vertex. As there can be several parents (and thus several names), we non-deterministically choose a parent v' of v along with the associated operation *op*:

$$\mathit{name}(v) = \begin{cases} \mathit{op}\big(\mathit{name}(v')\big) \text{ for some } (v', v, \mathit{op}) \in E & \text{if } \exists (v', v, \mathit{op}) \in E \\ \texttt{""} & \text{otherwise (root vertex)} \end{cases}$$

As an example, see how a name for $v_{f.val}$ is found: $\mathit{name}(v_{f.val}) = \mathit{op}_{f.val}(\mathit{name}(v_f)) = \mathit{op}_{f.val}(\mathit{op}_f(\texttt{""})) = \mathit{op}_{f.val}(\texttt{"f"}) = \texttt{"f.val"}$

For details on the construction of memory graphs from data structures, see Figure 8.

Fig. 5. The structure of memory graphs

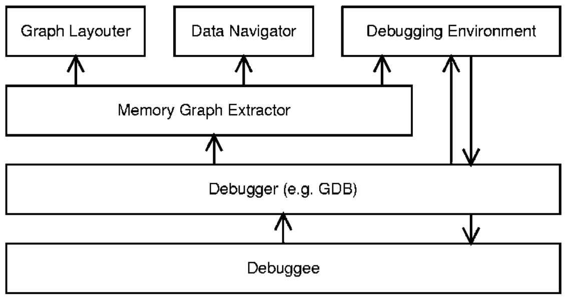


Fig. 6. Memory graphs within a debugging environment

```
 ...
(gdb) output &(pwd)
(passwd **) 0xbffff478
(gdb) output sizeof(pwd)
4
(gdb) output *(pwd)
{
  pw_name = 0x8057830 "zeller",
  pw_passwd = 0x8057837 "x",
  pw_uid = 501,
  pw_gid = 100,
  pw_gecos = 0x8057839 "Andreas Zeller",
  pw_dir = 0x8057848 "/home/zeller",
  pw_shell = 0x8057855 "/bin/bash"
}
```

```
(gdb) set variable $v17 = pwd
(gdb) output &((*$v17).pw_name)
(char **) 0x40133ce0
(gdb) output sizeof((*$v17).pw_name)
4
(gdb) output (*$v17).pw_name
0x8057830 "zeller"
(gdb) output strlen((*$v17).pw_name) + 1
7
(gdb) output &((*$v17).pw_passwd)
(char **) 0x40133ce4
(gdb) output sizeof((*$v17).pw_passwd)
4
(gdb)
 ...
```

Fig. 7. Dialogue between memory graph extractor and GDB

Slicing. A *memory slice*[2] is a subgraph of the memory graph. We distinguish two kinds of slices:

- A *backward slice* for a variable v consists of all the paths leading from the root vertex to v; it answers the question "On which paths can v possibly be influenced?".
  As a simple example, imagine you want to find out how a variable v can possibly be changed. The backward slice gives you all pointers referencing v.
- A *forward slice* consists of all the paths starting from v; it answers the question "On which paths can v possibly influence some other variable?"
  As another example, imagine you want to find out whether some part of memory becomes unreachable once a pointer v is changed. The forward slice tells you which variables were previously reached via v.

Slices are easily computed by determining reachable vertices.

Chopping. A *memory chop* between two variables v and w is the intersection of the forward slice of v and the backward slice of w; it answers the question "On which paths can v possibly influence w?".
If the chop of v and w is empty, then there is no way that v could possibly influence w. This is an important prerequisite for *garbage collection:* Whatever memory is to

[2] The name *slice* is adapted from the notion of *program slicing* on program dependency graphs [5].

Unfolding Data Structures

To obtain a memory graph $G = (V, E, root)$, as formalized in Figure 5, we use the following scheme:

1. Let *unfold*$(parent, op, G)$ be a procedure (sketched below) that takes the name of a parent expression *parent* and an operation *op* and unfolds the element $op(parent)$, adding new edges and vertices to the memory graph G.
2. Initialize $V = \{root\}$ and $E = \emptyset$.
3. For each base variable *name* in the program, invoke *unfold*$(root, \lambda x.\texttt{"}name\texttt{"})$.

The *unfold* procedure works as follows. Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let *tp* be the type of *expr*, and let *addr* be its address. The unfolding then depends on the structure of *expr*:

Aliases. If V already has a vertex v' at the same address and with the same type (formally, $\exists v' = (val', tp', addr') \in V \cdot tp = tp' \wedge addr = addr'$), do not unfold *expr* again; however, insert an edge $(parent, v', op)$ to the existing vertex.

As an example, consider the C statements:

```
struct foo f; int *p1; int *p2; p1 = p2 = &f;
```

If `f` has already been unfolded, we do not need to unfold its aliases `*p1` and `*p2`. However, we insert edges from `p1` and `p2` to `f`.

Records. Otherwise, if *expr* is a record containing n members $m_1, m_2, \ldots, m_n$, add a vertex $v = (\{\ldots\}, tp, addr)$ to V, and an edge $(parent, v, op)$ to E. For each $m_i \in \{m_1, m_2, \ldots, m_n\}$, invoke *unfold*$(expr, \lambda x.\texttt{"}x\texttt{.}m_i\texttt{"}, G)$, unfolding the record members.

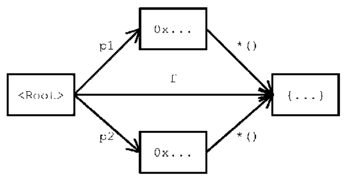
As an example, consider the "Edges" example in Figure 5. Here, the record `f` is created as a vertex and its member `f.val` has been unfolded.

Arrays. Otherwise, if *expr* is an array containing n members $m[0], m[1], \ldots, m[n-1]$, add a vertex $v = ([\ldots], tp, addr)$ to V, and an edge $(parent, v, op)$ to E. For each $i \in \{0, 1, \ldots, n\}$, invoke *unfold*$(expr, \lambda x.\texttt{"}x\texttt{[}i\texttt{]"}, G)$, unfolding the array elements.

Arrays are handled very much like records, so no example is given.

Pointers. Otherwise, if *expr* is a pointer with address value *val*, add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E. Invoke *unfold*$(expr, \lambda x.\texttt{"*(}x\texttt{)"}, G)$, unfolding the element *expr* points to (assuming that $*p$ is the dereferenced pointer p),

In the "Aliases" example above, we would end up with the following graph:



Atomic values. Otherwise, *expr* contains an atomic value *val*. Add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

As an example, see `f` in the figure above.

For more details on C structures, see Figure 9.

Fig. 8. The construction of memory graphs

Dealing with C data structures

In the programming language C, pointer accesses and type conversions are virtually unlimited, which makes extraction of data structures difficult. Here are some challenges and how we dealt with them.

Invalid pointers. In C, uninitialized pointers can contain arbitrary addresses. A pointer referencing invalid or uninitialized memory can quickly introduce lots of garbage into the memory graph.

To distinguish valid from invalid pointers, we use a *memory map*. Using debugger information, we detect individual memory areas like stack frames, heap areas requested via the *malloc* function, or static memory; a pointer is valid only if it points within a known area.

Dynamic arrays. In C, one can allocate arrays of arbitrary size on the heap via the *malloc* function. While the base address of the array is typically stored in a pointer, C offers no means to find out how many elements were actually allocated; keeping track of the size is left to the discretion of the programmer (and can thus not be inferred by us).

A similar case occurs when a C struct contains arrays that grow beyond its boundaries, as in `struct foo { int member; int array[1]; }` . Although `array` is declared to have only one element, it is actually used as dynamic array, expanding beyond the struct boundaries. Such structs are allocated such that there is sufficient space for both the struct and the desired number of array elements.

To determine the size of a dynamic array, we again use the memory map as described earlier: an array cannot cross the boundaries of its memory area. For instance, if we know the array lies within a memory area of 1000 bytes, the array cannot be longer than 1000 bytes.

Unions. The biggest obstacle in extracting data structures are C *unions*. Unions (also known as variant records) allow multiple types to be stored at the same memory address. Again, keeping track of the actual type is left to the discretion of the programmer; when extracting data structures, this information is not generally available.

To disambiguate unions, we employ a couple of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a *type tag*—an enumeration type within the enclosing struct whose value corresponds to the name of a union member. While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules—either hand-crafted or inferred from the program.

Strings. A `char` array in C has several usages: It can be used for strings, but is also frequently used as placeholder for other objects. For instance, the *malloc* function returns an `char` array of the desired size; it may be used for strings, but also for other objects.

Generally, we interpret `char` arrays as strings only if no other type claims the space. Thus, if a we have both a `char` array pointer and pointer of another type both pointing to the same area, we use the second pointer for unfolding.

Few of these problems exist in other programming languages. Most languages are far more unambiguous when it comes to interpreting memory contents; in object-oriented languages, unions are obsoleted by dynamic binding.

Fig. 9. Dealing with C data structures

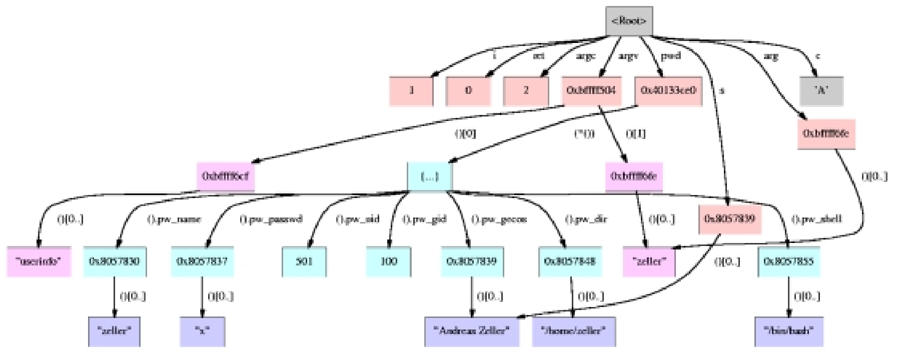
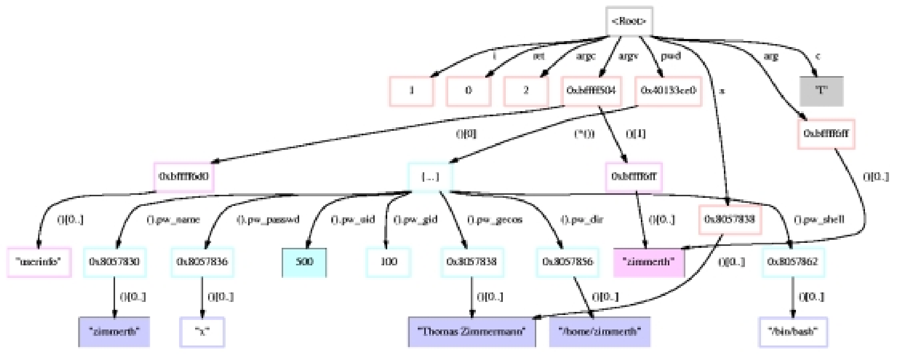



Fig. 10. Finding out what has changed

be collected must not be reachable from any base variable of the program. Using memory graphs, this property can easily be verified at run-time.

Searching. A feature not present in today's debuggers, but easy to realize using memory graphs, is to answer the question "Which variables all have a value of 1"? This can be used to "grep" the memory graph for specific values; the backward memory slices then tell the access paths to these values.
Searching is easily implemented using graph traversal.

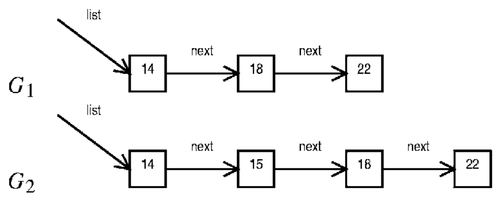
Clustering. To facilitate the comprehension and visualization of memory graphs, it may be useful to identify clusters in the graph and to condense them into single vertices. This can be done using graph information alone. It may be more effective, though, to cluster variables belonging into a specific module or package.
Clustering is an open issue for memory graphs; we expect most progress from domain-specific clustering methods.

5 Graph Differences

Another important application for memory graphs is *comparing program states*—that is, answering the question "What has changed between these two states?"

Comparing Memory Graphs

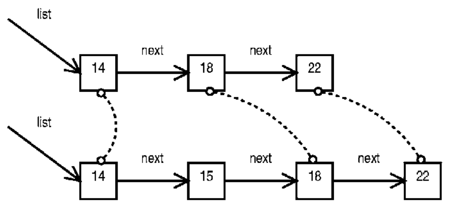
Since they abstract from concrete locations, memory graphs allow comparing program states on a *structural level*. As an example, consider these two memory graphs. What has changed?



As a human, you can quickly see that the element 15 has been inserted into the list. To detect this automatically, though, requires some graph operations. The basic idea is to compute a *maximum common subgraph* of G_1 and G_2 and to flag all the vertices that do not occur in both G_1 and G_2.
How does one compute a maximum common subgraph? Barrow and Burstall [1] first observed that a maximum subgraph can be obtained by using a *correspondence graph*. In our notation, their algorithm looks like this:

1. Create the set of all pairs of vertices (v_1, v_2) with the same value and the same type, one from each graph. Formally, $v_1 \in V_1$, $v_2 \in V_2$ and $val_1 = val_2 \wedge tp_1 = tp_2$ holds where $(val_1, tp_1, addr_1) = v_1$ and $(val_2, tp_2, addr_2) = v_2$.
2. Form the *correspondence graph* C whose nodes are the pairs from (1). Any two vertex pairs $v = (v_1, v_2)$ and $v' = (v'_1, v'_2)$ in C are connected if
   - the operations of the edges (v_1, v'_1, op_1) in G_1 and (v_2, v'_2, op_2) in G_2 are the same, i.e. $op_1 = op_2$, or
   - neither (v_1, v'_1, op_1) nor (v_2, v'_2, op_2) exist.
3. The maximal common subgraph then corresponds to the *maximum clique* in C—that is, a complete subgraph of C that is not contained in any other complete subgraph. This maximum clique can efficiently be computed using the algorithm of Bron and Kerbosch [2].

For our purposes, the resulting maximum clique (i.e. the set of corresponding vertices) already suffices: Any vertex that is not in the clique indicates a difference between G_1 and G_2.
The following figure shows the pairs obtained in (1). Since this is pretty unambiguous, finding the maximum clique is trivial—it is simply the one set of pairs. But it is plain to see that the element 15 in G_2 has no counterpart in G_1.



By highlighting inserted or deleted vertices this way, future debugging tools can quickly compare program states and identify what has changed between two states of a program run.

Fig. 11. Detecting differences

If the state is given in a name/value fashion, comparing states is difficult as soon as pointers come into play. For instance, we might like to invoke the *userinfo* program with a different UNIX user name. In this alternate run, all pointers can have different values (depending on the available memory), but still the same semantics. With a graph abstracting from concrete values, comparing program states becomes a rather simple graph operation—namely, the detection of the greatest common subgraph.

The construction details of the greatest common subgraph is described in Figure 11. In Figure 10, we see the result. The upper graph again shows the *userinfo* state, as in Figure 3. The lower graph shows the *userinfo* state when invoked with UNIX user name *zimmerth*; the common subgraph of the two graphs is outlined. One can clearly see the remaining differences.

If we knew, for instance, that the first run works fine, but the second does not, we know that the cause for the failure must be somewhere in the difference between the program state. Comparing memory graphs gives us this ability.

6 Querying Graph Structures

As a last memory graph, consider Figure 12. This memory graph was obtained from the GNU compiler as it compiled the C statement

```
z[i] = z[i] * (z[0] + 1.0);
```

The graph shows the statement as a *register transfer language* (RTL) tree, the internal representation of the intermediate language used by the GNU compiler. (The GNU compiler first converts its input into a syntax tree, which is transformed into RTL, which, after a series of optimizations, is then finalized into assembler language.)

This graph shows only a subset of the full GNU compiler state, whose memory graph at this time has about 40,000 vertices. However, even this subset is already close to the limits of visualization: if the RTL expression were any larger, we would no longer be able to depict it.

Nonetheless, we can use this graph to debug programs. It turns out that GCC crashes when its internal RTL expression takes this form. This is so because this RTL tree is not a tree; it contains a cycle in the lower right edge. This cycle causes an endless recursion in the GNU compiler, eventually eating up all available heap space.

We do not assume that programmers can spot cycles immediately from the visualization in Figure 12. However, we can imagine traditional graph properties (such as the graph being complete, cycle-free, its spanning tree having a the maximum depth and so forth) being computed for memory graphs, for instance in a debugging environment. A click on a button could identify the cycle and thus immediately point the programmer to the failure cause.

7 Drawing Memory Graphs

The figures in this paper were drawn in a straight-forward way using the DOT graph layouter from AT&T's *graphviz* package [3]. While these layouts are nice, they do not scale to large memory graphs (with 1,000 vertices and more).

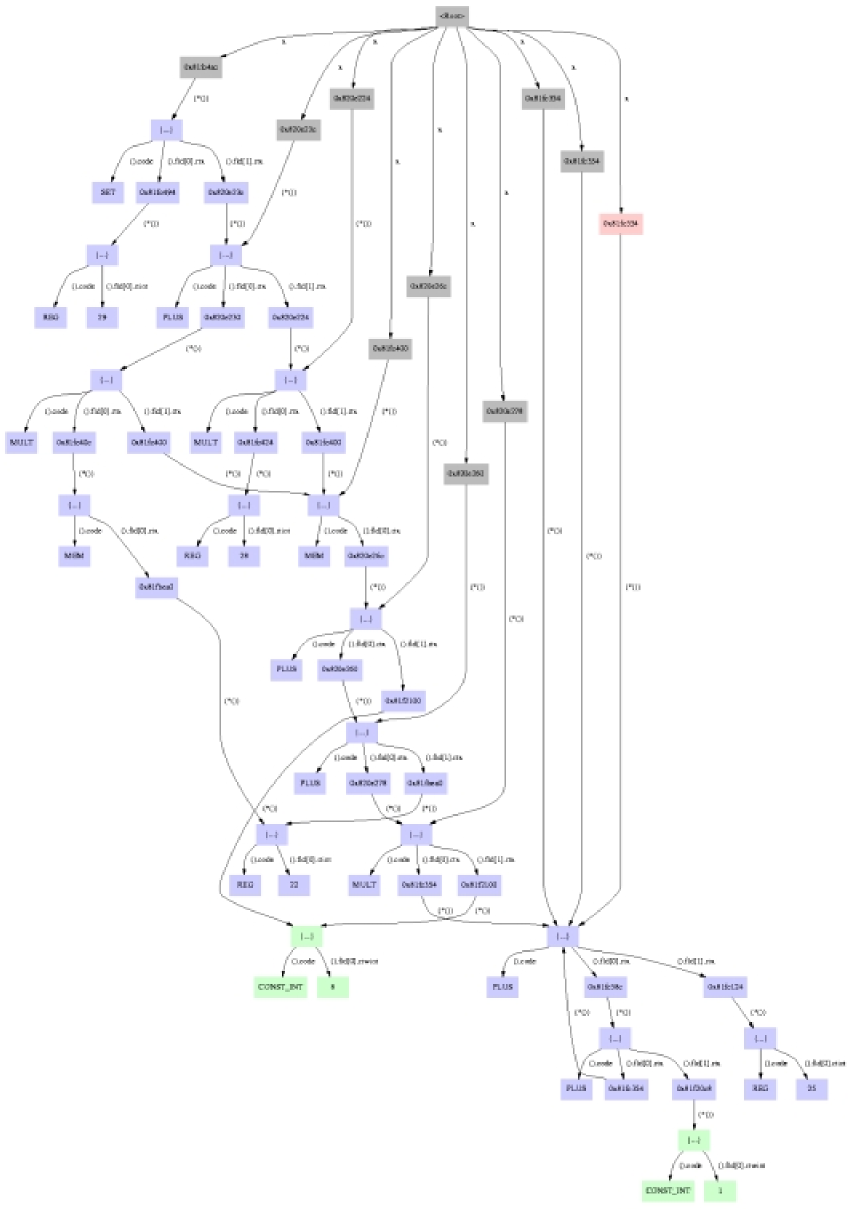


Fig. 12. An RTL tree in the GNU compiler. Can you spot the cycle?

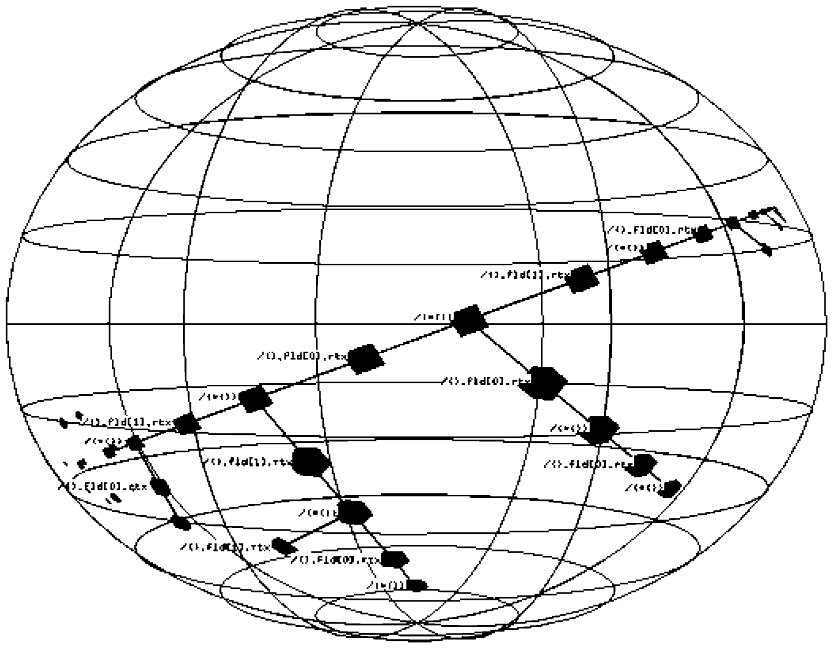


Fig. 13. The RTL tree from Figure 12 as visualized by *h3viewer*

More promising are *interactive* graph renderings that allow the user to navigate along the graph. We are currently experimenting with the *h3viewer* program that creates hyperbolic visualizations of very large graphs [4].

Figure 13 shows a screenshot of *h3viewer*.[3] The actual program is interactive: clicking on any vertex brings it to front, showing detailed information. By dragging and rotating the view, the programmer can quickly follow and examine data structures. If future successors to DDD will have an interactive graph drawing interface, it may look close to this.

Another idea to be explored for presentation is *summarizing* parts of the graph. For instance, rather than showing all n elements of a linked list, it may suffice to present only the basic *shape* of the list—in the style of *shape analysis* [6], for instance.

Finally, there are several pragmatic means to reduce the graph size: for instance, one can prune the graph at a certain depth, or, simpler still, restrict the view to a particular module or variable.

8 Conclusion and Future Work

Capturing memory states into a graph is new, and so are the applications on these graphs. Realizing appropriate navigation tools, efficient analysis and extraction methods and

[3] Colors have been altered to fit printing needs.

useful visual representations are challenges not without reward. Our future work will concentrate on the following topics:

Automated Debugging. By comparing two memory graphs—one of a working run and one of a failing run—and narrowing down the failure-inducing differences, as shown in [7], one may be able to isolate failure-inducing variable values during a program run. By repeating this at various places during program execution, one can isolate a whole *cause-effect chain* explaining how the failure came to be.

Queries. Once a memory graph is extracted, it may be useful to query the graph for specific properties (such as some subgraph being free of cycles, for instance). We are investigating into appropriate languages that allow us to specify and query such properties.

Visualization. Memory graphs are far larger than the classic examples used for validating graph drawing algorithms. Yet, memory graphs should have a far higher regularity than, say, random graphs. We are currently exploring how this regularity can be exploited to result in well-structured visualizations.

As a whole, memory graphs are an *enabling* technique that allow for new tools with new capabilities—be it for program analysis, program comprehension or automated debugging. Right now, we cannot yet see all the applications of memory graphs—and this is probably a good sign.

More information on memory graphs can be found at

`http://www.st.cs.uni-sb.de/memgraphs/`

References

1. H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, 1976.
2. C. Bron and J. Kerbosch. Algorithm 457–Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.
3. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.
4. T. Munzner. Drawing large graphs with h3viewer and site manager. In *Graph Drawing '98*, volume 1547 of *Lecture Notes in Computer Science*, pages 384–393, Montreal, Canada, Aug. 1998. Springer-Verlag.
5. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.
6. R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. of CC 2000: 9th International Conference on Compiler Construction*, Berlin, Germany, Mar. 2000.
7. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), Feb. 2002. To appear.
8. A. Zeller and D. Lütkehaus. DDD-A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, Jan. 1996.

Chapter 3
Software Visualization and Education

Introduction

John Domingue

Knowledge Media Institute,
The Open University, Walton Hall,
Milton Keynes, MK6 7AA, UK,
j.b.Domingue@open.ac.uk

Education has a distinguished history within SV. In fact, one of the most significant SV systems Sorting Out Sorting (SOS) [1, 2] was created to support students. SOS is a 30 minute film which describes nine sorting algorithms. The most referenced part of the film is one of the final scenes where a 'grand race' of the nine algorithms is depicted as nine simultaneously animated cartesian graphs. This scene, shown in figure 1, clearly indicates the different behaviour and performance of the nine algorithms. The impact of SOS was so profound that even today, over 20 years later, sorting is the most common domains used for demonstrating algorithm animation systems.

A second early piece of significant work was the computer lab at Brown University [3]. From 1983 algorithms and data structure courses have been taught in a classroom where each student has a dedicated networked workstation (initially Apollo, now Sun). Using the network instructors animating displays are copied to the students' screens. Although never formally evaluated anecdotal evidence indicated that the classroom is an extremely valuable resource for the students. The main problems encountered were associated with the effort which instructors had to put in to design, integrate and maintain the animations.

In the early days the main tool used with the computer lab at Brown was BALSA [4], which provided an elegant environment for programmers to create animations based on an extension of the classical graphics model-renderer paradigm.

1 Evaluation

At The Open University over two decades of work has been put into creating SV based environments for novice programmers. Most of this work has taken place within the context of a Cognitive Psychology course (D309). Within this course

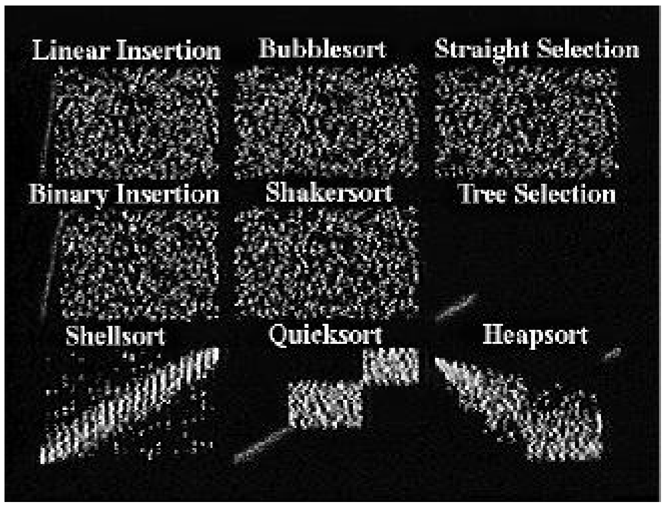


Fig. 1. The Grand Race scene from the Sorting Out Sorting video [included here by permission, Copyright 1997 ACM, Inc, originially published in Ron Baecker et al,"The Debugging Scandal and What to Do About It", Communications of the ACM, Vol. 40:4 (April 1997).]

students, who do not have a computing background, nor a computer, create programs which represent established cognitive models. Undergraduate courses at The Open University are taught at a distance with students attending a week long summer school each year. Most of the programming in this setting takes place in a two day period within the summer school. Early studies, [5, 6] indicated that whilst the majority of students could design algorithms, debugging even relatively small programs was a major hurdle. In the late 80s D309 moved from an in-house language to Prolog and a cradle-to-grave environment the Transparent Prolog Machine (TPM) [7, 8] was developed. In addition to the students on the D309 course, TPM was designed to support expert Prolog programmers (hence cradle-to-grave). These two communities had their own requirements. Students needed a clear and simple execution model whilst experts required powerful navigation techniques aligned to the ability to see low level details. TPM met these requirements by incorporating two tightly coupled AND/OR tree views at differing levels of abstraction. The coarse-grained view provided by TPM is shown below.

The feedback from the one thousand D309 students who took the course each year and from the expert Prolog programmers was positive and a masters course (DM862) was created which embedded the TPM notation throughout the textbook and video curriculum material [9, 10]. DM862 ran for 8 years attracting around 120 students each year.

In the early 90s TPM was evaluated against 3 textual Prolog tracers within the D309 course [11, 12]. Sixty four students participated in the experiment working in pairs to compare two similar programs. The students who used TPM took significantly longer to complete the experimental task than the students who used the

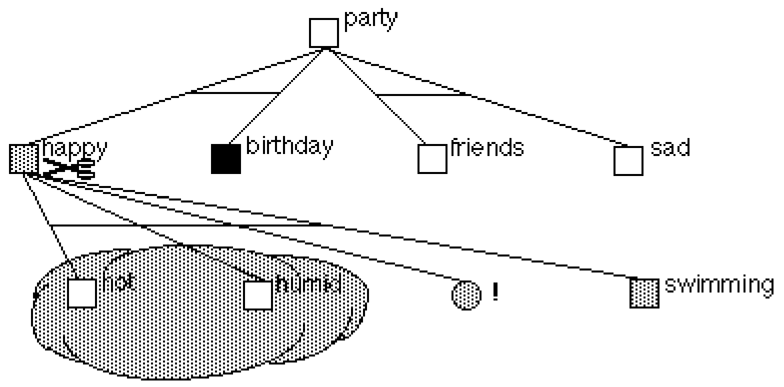


Fig. 2. TPM's coarse-grained view of Prolog execution. The scissors and the cloud indicate precisely the effect and scope of the Prolog cut primitive [taken by permission from http://kmi.open.ac.uk/kmi-misc/tpm/tpm.html]

textual tracers. From protocol analysis it was clear that when using TPM problems arose when a student had a misconception of the Prolog execution model. Instead of the visualization leading the student to correct their buggy execution model, TPM's graphical notation was re-interpreted by the student to match the incorrect model. An underlying cognitive model for this can be found in the work of Karmiloff-Smith [13, 14] in her studies of learning in children. She found that children move from simple "behavioural mastery" of domains to "deep understanding" through representational redescription. Representational redescription is a learning process where existing knowledge is unpacked and restructured to make key concepts and relationships explicit. Novice programmers, who only have behavioural mastery of a language, tend to map surface features of the code to surface features of the SV notation. A key requirement therefore for any SV used in a teaching context is that the notation must be explicit, thus avoiding ambiguous interpretation. Ideally, SV systems should use multiple notations to support learners as they progress from novices to expert programmers.

The above conclusions are (at least in part) backed up by a series of studies [15, 16] on the use of algorithm animations built in Tango [17, 18] and POLKA [19]. Summarising these results Stasko and Lawrence (in [16]) argue that when algorithm animations are used passively their impact on learning is minimal. The reason for this is that students do not know the algorithm that is being displayed, hence do not understand the operations or data objects and therefore can not make the mapping to their visual representation. This type of mapping problem occurred numerous times within their experiments. Their proposed solution is to engage students with the animation. Students should at least be able to enter their own input to the algorithm. Hundhausen and Douglas [20] conducted some interesting studies comparing the performance of students using POLKA against students who created their own algorithm animations using a selection of art supplies including paper, coloured pens, scissors and sticky tape. Unfortunately, no significant results were found. The results of this work, however, formed the design principles for SALSA, an interpreted, high-level language for constructing low fidelity algorithm animations and ALVIS a direct manipulation environment for SALSA. These are reported in the chapter by

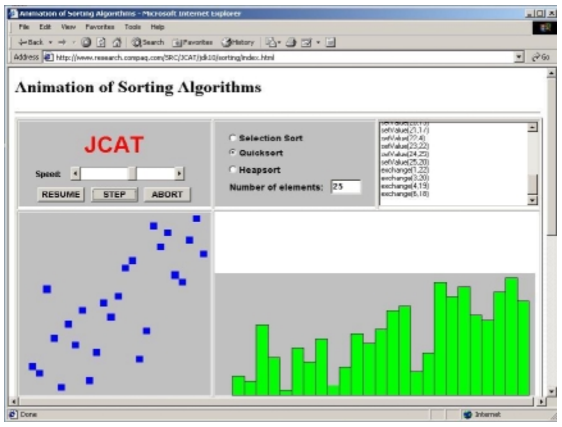
Hundhausen and Douglas *"A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations"*.
In *"Structure and Constraints in Interactive Exploratory Algorithm Learning"* Nils Faltin describes how students can engage with algorithm animations by breaking the algorithms into small chunks and encouraging the students to reconstruct the original algorithm. A somewhat similar approach is described *"Understanding Algorithms by Means of Visualized Path Testing"* by
Korhonen, Sutinen and Tarhio. In this chapter the authors outline a technique where algorithms are broken into blocks based on the algorithm pathways - pathways in the control flow graph of the algorithm.
Fleischer and Kucera use previous studies and systems to abstract out a number of design rules for constructing algorithm animations in *"Algorithm Animation for Teaching"*.
An extremely useful overview of 24 studies of the educational effectiveness of algorithm animations in education can be found in [21].

2 Web Based Environments

Since the mid 90s a number of web based SV systems specifically aimed at supporting learners have been created. The first system was Collaborative Active Textbooks (CAT) from Brown and Najork [22], which was web-based environment which allowed collaborative animations to be created and then be run simultaneously on a number of machines. The original CAT was written in Obliq [23] and required a specialist browser. A screen snapshot from the Java based version of CAT, JCAT [24], displaying a quicksort algorithm is shown in figure 3. The figure shows three views of the quicksort algorithm - for each animation, JCAT provides single-user, tutor and student views. Intended for classroom style teaching, the view and speed of student views can be controlled remotely using a tutor view. Recently, the ability to display 3D visualizations has been added to JCAT [25].

SAMBA [26] is a simple interface to POLKA [19], designed to enable computer science students to construct their own animations. JSAMBA [27] is a Java interface to SAMBA allowing animations to be constructed over the Web by typing commands into a text window within an applet. Each command consists of the command name (e.g. circle) followed by a unique identifier for the graphical object (e.g. 6). The remaining arguments specify the attributes of the graphical object. The commands define the appearance, location and movement of objects within the animation. The defined animation is then run in a separate browser window.

Another environment designed to enable users to easily create simple animations is Jeliot [28]. Animations of EJava (a specially created variant of Java) are created semi-automatically in Jeliot using a set of animated data types. Using a provided Java applet, users can send their code to a central server where the variables which can be animated are extracted before compilation by a standard Java compiler. Animation creation is based on a theatre metaphor. The compiled code is sent back and a director window is created. Animations are displayed on a stage which is controlled through a stage manager. The stage manager is used to select the variables to be animated and to specify their location and graphical representation. As the animation runs, the currently executing line of source code is highlighted. An example Jeliot animation is shown in figure 4. A simplified version of the system, Jeliot 2000, has recently been evaluated in class room setting with high school pupils [29].


Animation of Sorting Algorithms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.research.compaq.com/SRC/JCAT/jdk10/sorting/index.html
Go
Animation of Sorting Algorithms
JCAT
Speed:
RESUME
STEP
ABORT
Done
Internet

Selection Sort [✓ Quicksort] Heapsort

Number of elements: 25

```
setValue(21,17)
setValue(22,4)
setValue(23,22)
setValue(24,25)
setValue(25,20)
exchange(1,22)
exchange(3,20)
exchange(4,19)
exchange(6,18)
```

Fig. 3. A screen snapshot of JCAT displaying three views (dots, bars and code based) of the Quicksort Algorithm [taken by permission of Marc Najork and Compaq Systems Research Center from http://www.research.compaq.com/SRC/JCAT/jdk10/sorting/index.html]

The Internet Software Visualization Laboratory (ISVL) [30, 31] is a generic web based environment for displaying program visualizations over the web. Students can use ISVL to submit programs through a web browser. The programs are run on a central SV web server and a visualization is returned which can be played through using a video recorder style replay panel. The visualizations can be run in 'broadcast mode' whereby any students in 'receive mode' gets a live feed from the broadcasting machine. Asynchronous communication, an important component of distance education, is supported through the ability to annotate and store visualizations. Figure 5 shows a visualization of a student's (Bill) program which has subsequently been annotated by his tutor (Ingrid). ISVL also provides a facility for searching for visualizations by specifying visualization patterns. This facility effectively allows students to ask "has any other student had this problem before?".

In *"Hypertextbooks: Animated, Active Learning, Comprehensive Teaching and Learning Resources for the Web"* Ross describes a comprehensive set of web based learning resources incorporating built in SVs. Hypertextbooks allows students to follow threads of differing difficulty through the material using a ski slope metaphor.

In *"A taxonomy of network protocol visualization tools"* Crescenzi and Innocenti provide a taxonomy which aids educators in choosing network protocol visualization tools appropriate for a given teaching context.

As will become clear in the chapters in this section there has been a shift in recent years within the SV for education community. Today the design of SV frameworks

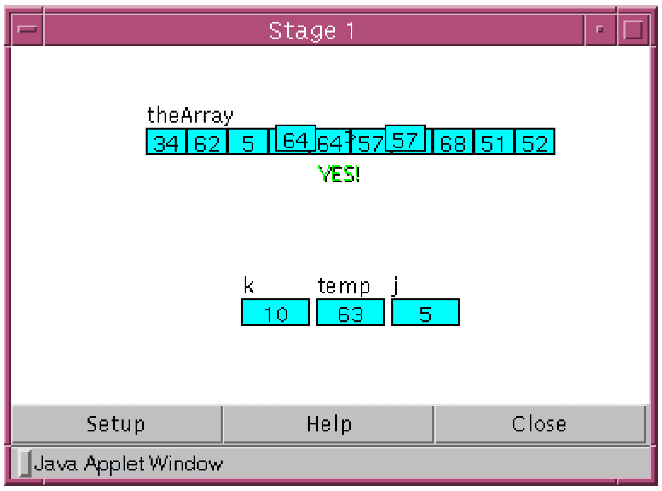
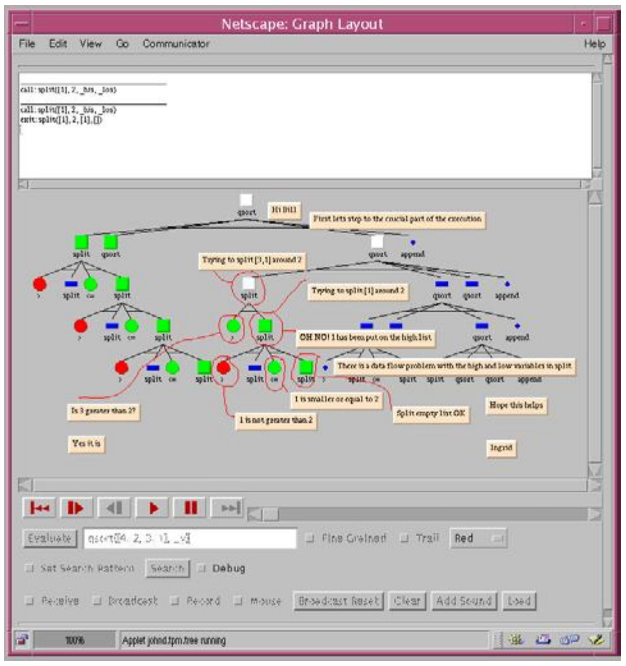


Fig. 4. A screen snapshot of a Jeliot animation of bubble sort. The array is shown at the top. K and J are loop counter variables and temp is a variable used to hold items in the array. The two array items 64 and 57 are being compared using the '>' operator (partially hidden by the 64 and 57 boxes).



```
call: split([1], 2, _his, _los)
call: split([1], 2, _his, _los)
exit: split([1], 2, [1], [])
```

Hi Bill
First lets step to the crucial part of the execution
Trying to split [1,1] around 2
Trying to split [1] around 2
OH NO! 1 has been put on the high list
There is a data flow problem with the high and low variables in split
1 is smaller or equal to 2
Split empty list OK
Hope this helps
Is 3 greater than 2?
1 is not greater than 2
Yes it is
Ingrid

qsort([4, 2, 3, 1], _X)

100% Applet johnd.tpm.tree running

Fig. 5. A screen snapshot showing a tutor Ingrid offering advice to a student Bill by annotating a visualization within ISVL.

and systems not only deploy state-of-the-art visualization techniques but use theories of learning, taken from the cognitive psychology community, to ensure that student activity results in significant learning outcomes. A future challenge is to create tools and methodologies which will result in the use of SVs by the majority of computer science educators.

References

1. Baecker, R.M., with the assistance of Dave Sherman Sorting out Sorting, colour sound film, University of Toronto. Distributed by Morgan Kaufmann. (1981)
2. Baecker, R.M. Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science in Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.) Software Visualization: Programming as a Multimedia Experience. MIT Press. (1998) 360-382
3. Bazik, J. Tamassia, R., Reiss, S., and van Dam, A. Software Visualization in Teaching at Brown University. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.) Software Visualization: Programming as a Multimedia Experience. MIT Press. (1998) 383-398
4. Brown, M. H. Algorithm Animation. ACM Distinguished Dissertations, MIT Press, New York. (1988)
5. Eisenstadt M., Breuker J. and Evertsz R., A cognitive account of “natural” looping constructs, Proceedings of the First IFIP Conference on Human-Computer Interaction, INTERACT ‘84, London, (1984) 173-177.
6. Eisenstadt M., Price B. and Domingue J., Software Visualization: Redressing ITS Fallacies, Proceedings of NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming, Genova, Italy (1992)
7. Eisenstadt M. and Brayshaw M., The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. Journal of Logic Programming, Vol. 5, No. 4, (1988) 277-342.
8. Brayshaw M. and Eisenstadt M., A Practical Tracer for Prolog International Journal of Man-Machine Studies, Vol. 35, No. 5 (1991) 597-631.
9. Eisenstadt M., Intensive Prolog. Associate Student Central Office (Course PD622), The Open University, Milton Keynes, UK: Open University Press, 1988.
10. Eisenstadt M. and Brayshaw M., A fine grained account of Prolog execution for teaching and debugging, Instructional Science, Vol. 19, No. 4/5, (1990) 407-436.
11. Mulholland, P. A Framework for Describing and Evaluating Software Visualization Systems: A Case-Study in Prolog, Ph.D. Thesis, The Knowledge Media Institute, The Open University (1995)
12. Mulholland, P. A Principled Approach to the Evaluation of SV: A Case Study in Prolog. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.) Software Visualization: Programming as a Multimedia Experience. MIT Press. (1998) 439-452
13. Karmiloff-Smith A., Micro- and macrodevelopmental changes in language acquisition and other representational systems. Cognitive Science, Vol. 3, (1979) 91-118.
14. Karmiloff-Smith A., Precis of Beyond Modularity: A developmental perspective on cognitive science. Behavioural and Brain Sciences, Vol. 17, No. 4, (1994) 693-745.
15. Stasko J., Badre A. and Lewis C., Do Algorithm Animations Assist Learning? an Empirical Study and Analysis. Proceedings of the INTERCHI ‘93 Conference on Human Factors in Computing Systems, Amsterdam, Netherlands, (1993) 61-66.
16. Stasko, J., and Lawrence, A. Empirically Assessing Algorithm Animations as Learning Aids. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.) Software Visualization: Programming as a Multimedia Experience. MIT Press. (1998) 419-438

17. Stasko J., The Path-Transition Paradigm: A Practical Methodology for Adding Animation To Program Interfaces, Journal of Visual Languages and Computing, Vol. 1, No. 3, September (1990) 213-236.
18. Stasko J., TANGO: A Framework and System for Algorithm Animation, Computer, Vol. 23, No. 9, September (1990) 27-39.
19. Stasko J. and Kraemer E., A Methodology for Building Application-Specific Visualizations of Parallel Programs, Journal of Parallel and Distributed Computing, Vol. 18, No. 2, June 1993, pp. 258-264.
20. Hundhausen, C. D. and Douglas, S.A. Using Visualizations to Learn Algorithms: Should Students Construct Their Own, or View an Expert's? In 2000 IEEE Symposium on Visual Languages Los Alamitos, CA: IEEE Computer Society Press, (2000) 21-28.
21. Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. A Meta-Study of Algorithm Visualization Effectiveness, In Eds. E. Sutinen and J. Domingue, Journal of Visual Languages and Computing Special Issue on Program Visualization, (2002 in press).
22. Brown, M. H. and Najork, M. A. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. Proceedings of the IEEE Symposium on Visual Languages (VL'96), Boulder, CO, Sept. 3-6, (1996) 266-275. Also available as Compaq Systems Research Center Research Report 142.
23. Cardelli, L. A language with distributed scope. Computing Systems 8, (1995) 27-59.
24. Brown, M. H. and Najork, M. A. Collaborative active textbooks. Journal of Visual Languages and Computing 8, (1997) 453-486.
25. Najork, M. A. and Brown, M. H. Three-Dimensional Web-Based Algorithm Animations. Research Report 170, Compaq Systems Research Center, Palo Alto, CA, July (2001).
26. Stasko, J. Using student-built algorithm animations as learning aids. Technical Report GIT-GVU-96-19 Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, US. (1996)
27. Stasko J. T. JSAMBA – Java version of the SAMBA animation program http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba/ (1997)
28. Haajanen, J. Pesonius, M. Sutinen, E., Tarhio, J., Teräsvirta, T. Vanninen, P. Animation of user algorithms on the Web. In: Proceedings of IEEE Symposium on Visual Languages (VL '97), IEEE. (1997) 360-367.
29. R. Ben-Bassat Levy, M. Ben-Ari, P.A. Uronen. An Extended Experiment with Jeliot 2000. Proceedings of the First Program Visualization Workshop. Porvoo, Finland, 2000, pp 131–140.
30. Domingue, J., and Mulholland, P. Fostering Debugging Communities on the Web. Communications of the ACM, Vol. 40, No. 4, April (1997) 65-71.
31. Domingue, J., and Mulholland, P. An Effective Web Based Software Visualization Learning Environment. Journal of Visual Languages and Computing.Vol 9 No. 5., October (1998) 485-508.

Structure and Constraints in Interactive Exploratory Algorithm Learning

Nils Faltin

C. v. Ossietzky Universität Oldenburg
Abt. Computer Graphics & Software-Ergonomie
D – 26111 Oldenburg, Germany
http://faltin.de
Faltin@Informatik.Uni-Oldenburg.DE

Abstract. Traditionally an algorithm is taught by presenting and explaining the problem, the algorithm pseudocode and an algorithm animation or a sequence of static snapshots. My aim is to foster creativity, motivation and high level programming concepts by providing the student an alternative route to algorithm understanding: exploratory learning. The algorithm is structured into several functions and this structure is presented to the student. The student is encouraged to device a pseudocode description himself. An instance of the problem is presented on the level of each algorithm function. A graphical simulation of the data structures and some of the algorithm functions are provided. It is the student's task to find out a correct sequence of function calls that will solve the problem instance. The instructor can control the difficulty of the task by providing algorithm constraints. Each new constraint will shrink the solution space and thus ease the task.

1 Introduction

1.1 Why Are Algorithm Animations Not Used Widely?

Algorithms and data structures are an important part of computer science education. Textbooks and instructors routinely use graphical representations of the data structure to explain the working of an algorithm. Because an algorithm changes its data structure over time, it seems natural to portray this process using a film (an algorithm animation). The field of algorithm animation has mainly been working on technical support for the production of such films. The book by Stasko et al. gives a good overview on algorithm animation [1]. Although it has become easier to produce animations and a lot of animations are available on the internet, it seems that only few instructors and text book authors use animations to explain algorithms. This problem is clearly recognized by the community. In the motivation to the Dagstuhl seminar on Software Visualized in May 2001 the participants were asked to answer the question: „Why is software visualization not widely used in education and software development?“.

1.2 Animations Must First Become Effective by Promoting Active Learning

Many animations available today only allow the student to passively watch the animation.The reluctancy of some computer science instructors to use algorithm animations may well be based on their intuition that such animations are not more effective than conventional media like transparencies or blackboards. They feel that students should be more actively involved in the learning process. This view is supported by a new meta-study of 24 controlled experiments evaluating algorithm animation effectiveness [2]. Most of the experiments evaluate the effect of different types of learner involvement. One type, passive learning, encompasses learners reading textual material, listening to a lecture or passively viewing a visualization. The other type, active learning, encompasses learners answering questions, constructing input data sets, predicting algorithm steps, programming an algorithm, or constructing an algorithm animation. The experiments showed a clear advantage of active over passive learning. Of the 14 experiments that compared levels of learner involvement, 10 showed a significant advantage of active learning, no experiment showed an advantage of passive learning and 4 experiments did not show a significant difference.

1.3 Examples for Approaches to Active Learning

A number of research projects have developed methods and software for engaging computer science students in active learning of algorithms. The following list gives some examples of such approaches. Four more examples can be found in the meta-study ([2], sec. 5.1.4).

Social learning: Christopher Hundhausen proposes a constructivistic view of learning in which the student takes on increasingly central roles within a community of active professionals [3]. Algorithm animations are constructed by small student groups and are presented and discussed with their peers and the instructor. The animations can be made out of simple art supplies or with a special authoring tool. The ALVIS/SALSA tool he developed supports direct manipulation of graphical primitives (like a drawing program), self-displaying objects, relative object coordinates, a scripting language, flexible execution control and on-the-fly changing of an animation during presentation. The tool is described in detail in a paper by Hundhausen in this volume. Students carefully choose a small set of interesting input data and show the algorithm executing on it. There is no need to program an algorithm animation for general input. He studied this teaching method in an undergraduate course and documented it with ethnographic methods. The instructor and the students liked the teaching method and benefited from it.

Interactive construction: Susan Rodger developed the JFLAP system for an automata theory course [4]. It allows students to create and simulate automata and to convert between automata, grammars and regular expressions. Students can convert the automata manually in stages with ‚help‘ or watch an animation. JFLAP can be used during lecture, in a lab or at the students‘ home.

Predicting algorithm steps: Several systems have been built that allow an animation to run in self-test mode. In this mode the animation pauses at certain points and asks the student to predict the next step of the algorithm. He may be asked to click on data elements or type in numerical values. For example the heapsort algorithm can be run in self-test mode in the system ‚Algorithms in Action' [5].

Visual programming: Amir Michail built the OPSIS system for visual programming of algorithms that work on binary search trees [6]. A program is a kind of finite state machine. A state corresponds to a set of states of the data structure that conform to a constraint. The graph of program states can be viewed as a visual proof for the correctness of the program. The student programs by selecting a state and assigning an operation to the state. The system will automatically generate one or more successor states. Loops are formed by identifying a successor state with an already existing state. The program can be executed on input data and the program trace will be visualized.

Exploratory learning: I have developed a method for interactive exploratory algorithm learning. Students re-invent parts of an algorithm. They work with a series of interactive algorithm simulations that allow them to experiment with algorithm functions and try out their solutions. The method is described in detail in ([17], [18]) and briefly sketched in sections 3.2 to 3.6.

1.4 Overview of Following Sections

I will go into detail about some concepts that underlie my method of exploratory learning, but that can also be used independently. Section 2: *Functional structuring* means to divide an algorithm into many small functions that can be explained and comprehended independently. Section 3: Instead of giving all information about an algorithm, as in a typical lecture or textbook, students should have the opportunity and the environment to find out some aspects themselves (*exploratory learning*). Section 4: *Constraints* and invariants are commonly used to proof the correctness of an algorithm, but they can also be used to describe an algorithm and to control the difficulty of exploratory learning tasks.

2 Functional Structure

It is a basic heuristic principle to structure systems recursively into modules in order to manage their complexity. The structuring principle is used in several domains that are linked to algorithm learning and where comprehensibility is a main goal:

- Research in cognitive psychology suggests that knowledge in the brain is organized as a net of concepts ([8], pp 144-147; [9], chapter 5).

- Physical skills like juggling and stilt walking can be structured into subskills that can be named and trained separately. Seymour Papert reports that the time needed to learn juggling is reduced from several hours to about half an hour when subskills are taught separately and not as a sequence of basic steps. Learning structured programming and debugging in LOGO enabled children to better describe their experiences learning stilt walking ([10], chapter 4).
- Software engineering suggests to construct software out of interrelated modules. Modularization is a main theme in Bertrand Meyers textbook ([7], chapter 3).
- Textbooks typically explain an algorithm in a structured way. The different aspects of the data structure are shown. The code is divided into several functions that are explained individually. For example the heapsort algorithm is divided into six functions in the Cormen et al. textbook ([13], chapter 7). Sedgewick uses only two functions in his textbook ([14], chapter 11), but explains the build-heap and sorting phase in the text, which can be viewed as a replacement for defining functions.

But structuring does not come without cost. Dividing an algorithm into several functions introduces function declarations and function calls into the code. This makes the code longer and can slow down execution, if the compiler does not inline the called functions. And some minor performance optimization opportunities may be lost. We see a low-structured style in the original publication of heapsort [15] which uses only two heapsort-specific functions (besides an exchange function) and extends the build-heap phase into the first pass of the sort phase. It seems that the need for a short description in the journal and fast execution on a computer were the motivation for using a low-structured style.

While this answer to the trade-off between presentation space and comprehensibility may be acceptable for a scientific publication it is certainly not appropriate when teaching algorithms to students. Unfortunately most authors of algorithm animations follow a low-structured style and only create one animation that shows the basic steps of an algorithm. As an example, most animations of quicksort just show the comparing and exchanging of elements, but not the main functions quicksort and partition. It would be better to create a separate animation for each algorithm function and visualize each call to a helper function as one interesting event [16].

Structuring an Algorithm into functions is a prerequisite for the design of algorithm simulations, that are described below.

2.1 Structured Programming - A LOGO Example

Seymour Papert gives an example for the benefits of structured programming in the LOGO programming environment ([10], pp. 100-103). A child had chosen to draw a stick figure of a man (fig. 1, adapted from [10], p. 100) with the LOGO turtle. In his first, unstructured approach he mimiced a paper-based dot-to-dot drawing. He programmed the drawing by a sequence of turtle commands in a procedure man: turn the turtle, draw a line for the left leg, turn, move back, turn, draw the right leg, turn, move back and so on (fig. 2). For the 10 dots this leads to about 25 turtle commands. But the resulting drawing did not resemble the desired stick figure. It was very hard to find the bug in this one large procedure. This experience led the child to appreciate structured programming.

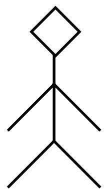


Fig. 1. Stick figure man

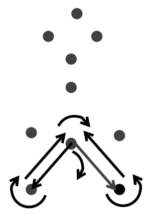


Fig. 2. Dot-to-dot

Papert then shows how the stick figure can be drawn with a structured LOGO program. The program consists of the procedures man, wedge, head, line and square. Each procedure draws a geometric part of the figure. The man procedure calls wedge and head, while wedge calls line and head calls square (fig. 3). The LOGO code for the procedure man is shown in fig. 5 (adapted from [10], p. 102). Such a structured program is much easier to understand, because each procedure can be understood and tested separately.

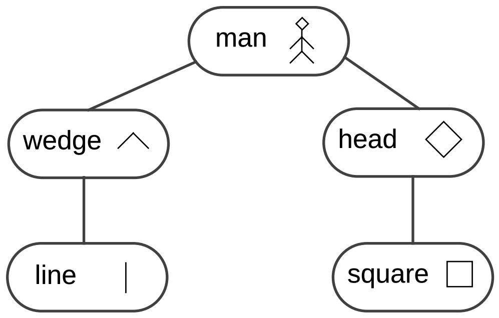


Fig. 3. Function call

3 Exploratory Algorithm Learning

3.1 Learning Geometry in Mathland

Children learn their mother tongue easily in their first years of life, without receiving any formal education. Even learning a second language is easy for a child, if it lives in a culture that speaks that language. So if learning French is best done in France, what is the best place to learn geometry in? It was the vision of Seymour Papert to develop a ‚mathland‘, in which it should be natural, easy and enjoyable to learn geometry [10]. He saw the computer as a new educational tool that could serve as a medium for access to mathland. In Papert‘s mathland the child programmed a mechanical robot, the turtle, to plot drawings on paper or a screen turtle to produce colored drawings on the computer screen. The children worked on small self-chosen programming projects. They interacted with the turtle through the textual programming language LOGO. The work was motivating, because they pursued their own goals and received visual feedback through the generated drawings. Their experiences in the turtle environment led them to appreciate heuristic strategies like structure and debugging and domain-specific theorems. An example is the turtle round-trip theorem that states that a turtle that draws the border of a shape will in sum make a 360 degrees rotation.

The LOGO turtle is an example of an *exploratory learning environment* for the domain of geometry. Much of the motivation, insight and knowledge does not come from teaching but from own experiences in the learning environment. It is the child that is actively exploring the rules and concepts of the learning environment. The task of the teacher changes from instructing to preparing the environment, giving a short introduction to it and helping the child when it gets stuck.

3.2 Learning an Algorithm Function with a Simulation

I have developed a framework for courseware that realizes an exploratory learning environments for algorithms ([17], [18]). Two examples for such courseware are described in section 3.4. Hansen, Narayanan and Schrimpsher have developed a similar framework for embedding algorithm animations in hypermedia environments [11]. Their emphasis is on active learning and on presenting multiple views and approaches to the algorithm. They do not promote exploratory learning. Similar to [5] they do present an algorithm on different levels corresponding to pseudocode segments, but they do not work with a strict functional structure as is done in my approach.

According to my approach, an author of a learning environment shall decompose an algorithm into many small functions and draw a call graph for the functions. The call graph is shown to the student and the data structure and the purpose of each function is explained. It is the student's task to discover the correct algorithm steps himself. To aid him an interactive visual simulation is offered for each function of the algorithm. It shows the data structure and a button for the functions that are called from the actual function (Fig. 4). The interactive visualization serves as the user interface to the callable algorithm functions and the data structure. The student is free to perform any sequence of function calls that reaches the goal of the current algorithm function. He is not bound to perform the exact steps of the actual algorithm, as he is with the ‚predict the next step' type systems like [5]. The student performs a step by clicking on objects in the data structure and starting functions by clicking on the buttons. In some sense the student is acting as the computer at the level of the current function. Past empirical research points out the importance of a step back/ rewind feature in algorithm animations [12]. If the student realizes that he has done steps in error, he should be able to undo them. So the algorithm simulation must both simulate the function calls and provide mechanisms for undo. At any time the student can consult a tutorial text explaining the standard solution and watch an animation of the actual algorithm.

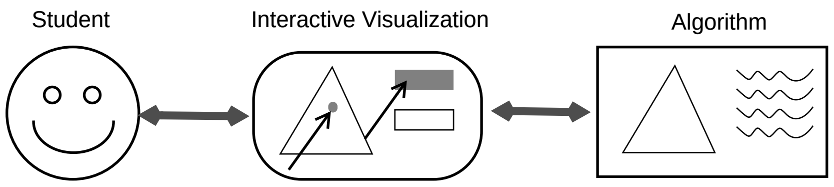


Fig. 4. Exploring an Algorithm function

```
TO man
   wedge
   FORWARD 50
   wedge
   FORWARD 25
   head
END
```

Fig. 5. Function man in LOGO

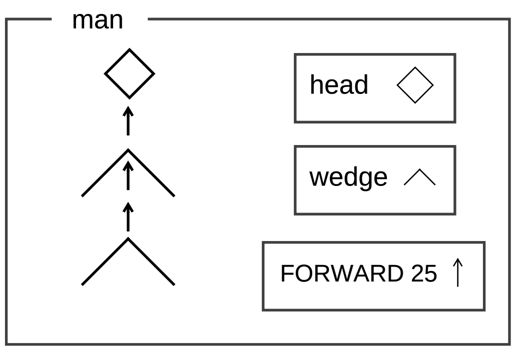


Fig. 6. Algorithm simulation for man function

3.3 An Algorithm Simulation for the LOGO Function Man

If we view the stick figure program as an algorithm we would provide an interactive simulation for each of the procedures man, wedge, head, line and square. Fig. 6 shows an algorithm simulation for the man function. The goal for the student is to draw the complete stick figure. The procedures wedge and head can be called by clicking on the corresponding button. The „Forward 25" button is provided to simulate an execution of the basic LOGO command Forward with parameter 25, which moves the turtle forward 25 distance units and draws a line. Note that in this example, there is no need to provide actual parameters for the function calls. It suffices to click on the buttons.

Initially the drawing space left of the buttons contains only an arrowhead marking the position and heading of the turtle. By pressing the wedge button the legs are drawn. The correct solution is to continue with two times Forward 25, wedge, Forward 25 and head. For each button pressed, the simulation responds by drawing the corresponding figure. At the end, the complete stick figure is drawn, as shown in fig. 6.

3.4 Courseware for Exploratory Learning

Our research group has developed webbased courseware on the algorithms following my guidelines for exploratory learning. The courseware on Binomial Heap (fig. 7) and Heapsort (fig. 8) [19] are freely available over the internet under the GNU General Public License. They contain explaining text and figures, like those found in a textbook and

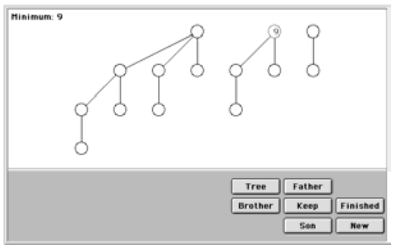
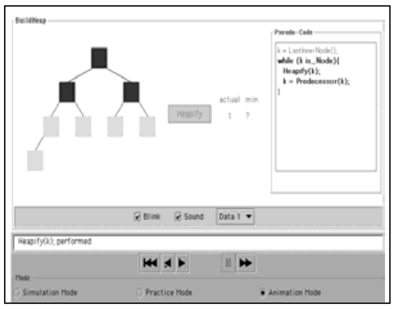


Fig. 7. Binomial Heap: Find Minimum



```
k = LastInnerNode();
while (k is_Node){
  Heapify(k);
  k = Predecessor(k);
}
```

Heapify(k); performed

Fig. 8. Heapsort: Build Heap

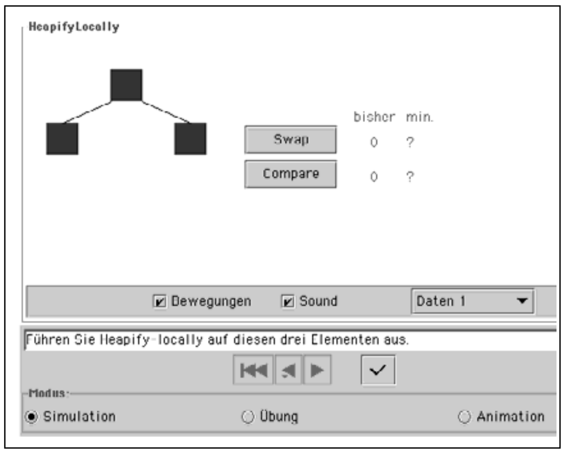
several interactive simulations of algorithm functions as Java applets. The binomial heap courseware has a comprehensive tutorial text. Some of the applets allow several paths to the solution, while some are restricted to one solution path. The tutorial text is currently only available in german, while the applets are available in german and english. The courseware on Heapsort has less tutorial text. Its applets can be used in simulation, practicing and animation mode. In simulation mode the student performs the algorithm steps manually and all solution paths are accepted. In practicing mode only the solution path of the standard algorithm is accepted. But now the student can request hints to complete the task. In animation mode, the student controls the animation with a VCR-like panel and all steps are performed automatically. The pseudocode of the function is displayed with the active line highlighted. All modes support multiple undo/step backward and smooth visual transitions between states of the data structure. The applets use our SALABIM java library to manage undo information, to check user steps against standard solution steps and to realize the user interface of execution control. The heapsort courseware is available in english and german.

3.5 Evaluating the Courseware

The courseware on Binomial Heap was formally evaluated in five sessions each with one computer science student. The student was watched while he was reading the courseware pages and working with the applets. This uncovered some usability problems, like for example a link that led several students astray. After that students answered a test asking procedural and conceptual questions about the algorithm. Four of the five students performed very well in the test. Finally the students were interviewed and asked about their learning style, opinion and preferences. This evaluation helped to identify and fix problems in the courseware.

Both the courseware on Binomial Heap and on Heapsort were used in a first year computer science course on algorithms and data structures at our university of Oldenburg, germany. Students received assignments to work through the courseware and then work on some traditional paper-and-pen or programming task. Many students had technical problems with the heapsort java applets, because they had to install the Java-Swing library first. So we changed the courseware, so that the library will be loaded automatically over the Internet, even if this takes longer time than with a locally installed library. The majority of students had no technical problems with the Binomial Heap courseware.

Most students worked in groups of two on the assignments. Student work groups, teaching assistants and the instructor were asked to rate both courseware products on a questionnaire on a scale from one to six (1 = „very helpful for learning the algorithm", 2= „helpful", 3= „useful", 4= „questionable whether helpful", 5= „useless", 6= „harms learning"). The heapsort courseware received an average rating of 2,64 (with 44 questionnaires returned) and Binomial Heap a rating of 2.05 (40 questionnaires returned).



[✓] Dewegungen [✓] Sound

Führen Sie Heapify-locally auf diesen drei Elementen aus.

[✓ Simulation] Übung Animation

Fig. 9. Simulation of Heapify-Locally from Heapsort

3.6 An Algorithm Simulation Example

Figure 9 shows the simulation of the function Heapify-Locally from the courseware on heapsort. The student has the task to establish the heap order in the three nodes heap tree. The heap order demands that the key in a parent node must be greater or equal than the keys in the child nodes. At the beginning of the exercise the content of the nodes is unknown, so they are displayed in dark grey. Users compare two nodes by clicking on the nodes and then on the button „Compare", which makes the keys visible and displays a relation sign (<, =, >) between the nodes. Keys are visualized by squares proportional in area to the key size. Users swap the keys of two nodes with the button „Swap".

Figure 10 shows the steps of the standard solution from the actual heapsort algorithm. First both child nodes are compared. The larger child node is compared with the parent node. If necessary the keys of the parent and the larger child node are swapped.

To enable exploratory learning, the simulation allows arbitrary step sequences. So the student can try different solutions to find an optimal solution. Figure 11 shows a different and longer step sequence to reach the goal of establishing the heap order. The parent node is compared with the left child. Because the child contains a larger key, keys are swapped. Likewise the parent node is compared with the right child node and again keys are swapped.

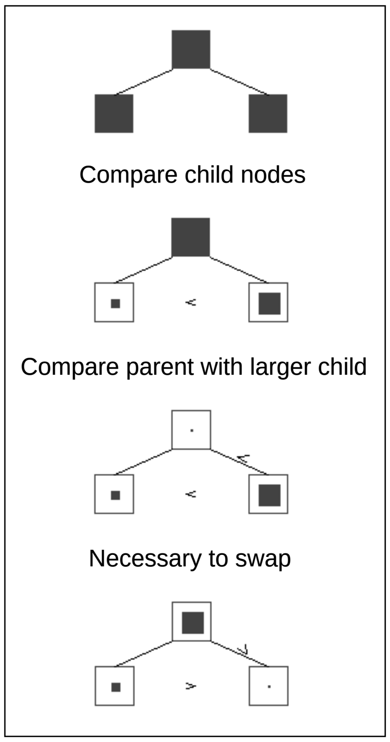


Fig. 10. Standard solution for Heapify-Locally

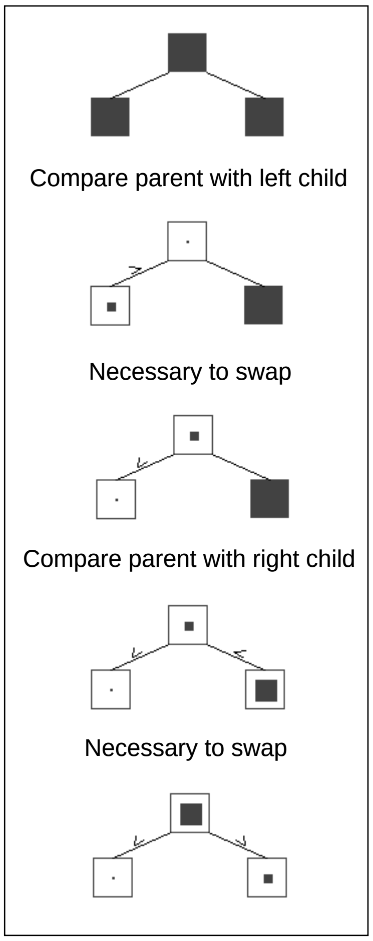

Compare parent with left child
Necessary to swap
Compare parent with right child
Necessary to swap

Fig. 11. Alternative solution for Heapify-Locally

4 Constraints Delimit Exploration Space

An algorithm simulation can be viewed as a user interface to an exploration space. An exploration space is a graph with vertices corresponding to function calls and edges corresponding to states of the data structure. The algorithm simulation visualizes a state of the data structure and waits for user input. The user selects actual parameters in the data structure and calls a function by pressing a button. This will change the data structure which corresponds to following a vertex in the exploration space graph. The change may be visualized by a smooth animation to the new state.

In the algorithm simulation for the man function, each of the functions head, wedge and forward can be called at any time. This corresponds to an exploration space with infinitely many edges and vertices. While the solution is easy to find in the example of the man function, it may be very difficult to find a path to the solution for more realistic algorithm functions. Thus an author of an algorithm simulation needs a systematic way to control the difficulty of the students task to find a path. She can do so by adding rules that disallow some of the vertices. These rules are called constraints. The algorithm invariants are an important class of constraints. The more constraints are provided, the smaller the exploration space gets. When there is only one chain of edges left, she has in fact specified the algorithm completely. Adding more constraints would render the task unsolvable.

4.1 Constraints and the Breadth First Graph Traversal

I will use the example of the breadth first graph traversal algorithm to explain the constraints concept. The algorithm solves the traversal problem: Given a graph with nodes and connections and one start node visit all nodes in the graph (fig. 12).

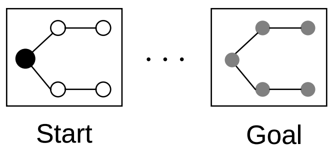


Fig. 12. Traversal Problem

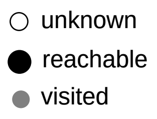


Fig. 13. Node States

The algorithm can be specified by stating the following four constraints:

1) Reachable Neighbor Constraint

The reachable neighbor constraint specifies that only neighbors of already visited nodes are reachable. This is encoded in the visual classification of nodes (fig. 13). Nodes which the algorithm does not know of yet are drawn as a small circle with a white interior. When a node becomes reachable, it is drawn with a large black disc to indicate that it can be clicked on to be visited. These nodes are the „working set“ of the algorithm. Nodes already visited need not be considered again, so they are drawn as small grey discs.

2) Sequential Constraint

It is assumed that the algorithm runs on a sequential computer so it can only visit one node at a time, even if several are reachable.

Neighbor and Sequential constraint together form an exploration space shown in fig. 14. Note that unlike the general case for algorithm simulations there are no „dead ends“ in this example. Every path eventually leads to the solution state. Note also, that there are several equally legitimate „solution paths“ to reach the goal. The „algorithm“ is still some kind of „non-deterministic“.

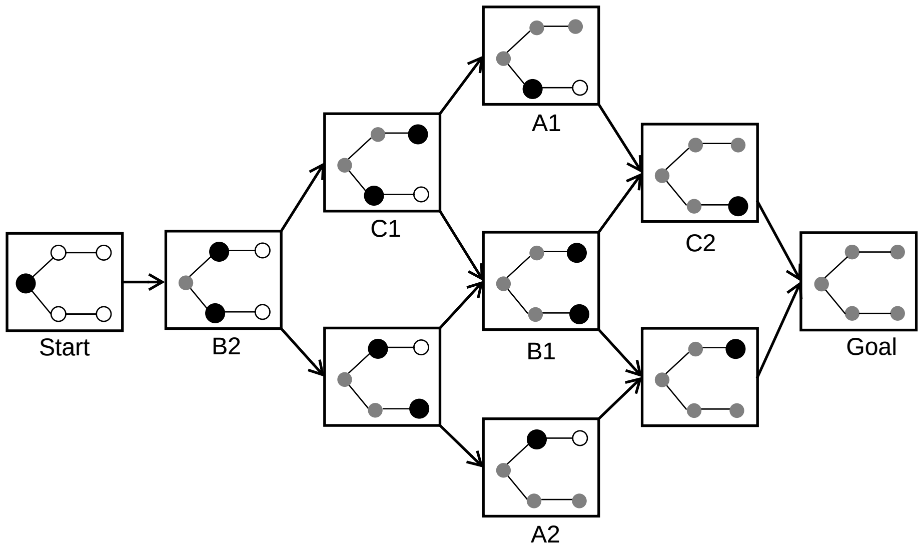


Fig. 14. Exploration space for graph traversal

3) Breadth First Constraint

Up to now the student had the liberty to click on any of the reachable nodes. Now this liberty is reduced significantly. In an implementation, the algorithm has to store the reachable nodes. The constraint demands that they are stored in a FIFO queue.

This removes the edges A1 and A2 from the exploration space, leaving two edges (B1 and B2) with multiple successors.

If we had demanded the use of a LIFO Stack, the edge B1 would have been removed, resulting in a depth first algorithm.

4) Neighbor Sequence Constraint

When a reachable node is selected, its yet-unknown neighbors become reachable. The neighbor sequence constraint establishes a sequence among these newly reachable nodes. It comes from the fact that in an implementation of a graph there is typically some built-in sequence at which neighbors of a node can be accessed. The constraint enforces one such sequence: these nodes can only be visited in a sequence starting with the lowest node and continuing counterclockwise. Of course other sequence rules are possible.

This constraint removes the edges C1 and C2. Now there is only one chain of edges and vertices left. It starts at the start vertex and ends at the goal vertex. This corresponds to a trace of a classic (deterministic) implementation of a breadth first traversal.

5 Conclusion

Research in learning effectiveness of algorithm animations suggest that students should participate actively in the learning process instead of just watching an animation. My approach is to provide the student with an algorithm simulation as an environment for exploratory learning, so that he actively reinvents parts of an algorithm. To make that task feasible, an algorithm is structured into many small functions so the student only has to think about one function at a time. Finding the steps of a function for a specific data input can further be eased by providing constraints that delimit the exploration space.

For the future of algorithms teaching I expect a more widespread use of active learning approaches. The approaches enumerated in section 1.3 are not mutually excluding each other. It is worth considering a careful combination of these for future learning scenarios. For example students could develop prototypes of exploratory learning environments for an algorithm using tools like ALVIS/SALSA, discuss them with their peers and the instructor and then implement them as Java applets.

References

[1] John T. Stasko, John B. Domingue, Marc H. Brown und Blaine A. Price (Hrsg.): *„Software Visualization“*. 1998. MIT Press, Cambridge, Massachusetts.

[2] Christopher Hundhausen, Sarah Douglas and John Stasko. (In press). A meta-study of algorithm visualization effectiveness. Journal of Visual Languages and Computing, to appear.

[3] Christopher Hundhausen: *„Toward Effective Algorithm Visualization Artifacts. Designing for Participation and Communication in an Undergraduate Algorithms Course“*. Dissertation. June 1999. CIS-TR-99-07. Dept. of Comp. and. Inf. Science, University of Oregon, Eugene, USA. http://lilt.ics.hawaii.edu/~hundhaus/dis/.

[4] T. Hung and S. H. Rodger: „ *Increasing Visualization and Interaction in the Automata Theory Course*“, Thirty-first SIGCSE Technical Symposium on Computer Science Education, p. 6-10, 2000.

[5] Linda Stern, Harald Sondergaard and Lee Naish: *„A Strategy for Managing Content Complexity in Algorithm Animation“*. PP 127-130 in: Bill Manaris (Ed.), Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education - ITiCSE '99. June 1999. ACM Press, New York.

[6] Amir Michail: „*Teaching Binary Tree Algorithms though Visual Programming*“. In Symposium on Visual Languages, pages 38--45, IEEE, September 1996. http://opsis.sourceforge.net/

[7] Bertrand Meyer: „Object-Oriented *Software Construction*„. 2nd ed. 1997. Prentice Hall.

[8] John R. Anderson: „Kognitive *Psychologie*„. 2. ed.. 1996. Spektrum Akad. Verlag. Translated from „Cognitive Psychology and its Implications“. 4th ed.. 1995. Freeman.

[9] Walter Edelmann: „Lernpsychologie“. 5. Auflage. 1996. Psychologie Verlags Union.

[10] Seymour Papert: „Mindstorms. *Children Computers and Powerful Ideas.„*. 1980. Basic Books.

[11] Steven R. Hansen, N. Hari Narayanan and Dan Schrimpsher. 2000. Helping learners visualize and comprehend algorithms. Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, 1(1).http://imej.wfu.edu/articles/2000/1/02/index.asp

[12] John Stasko, Albert Badre and C. Lewis. Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. 1993. In Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems (pp. 61-66). New York: ACM Press.

[13] Thomas H. Cormen, Charles E. Leiserson und Ronald L. Rivest: „Introduction *to Algorithms„*. 1990. MIT Press.

[14] Robert Sedgewick: „Algorithms *in C++„*. 1992. Addison-Wesley.

[15] Robert W. Floyd. *Algorithm 245 (treesort)*. Communications of the ACM, 7:701, 1964.

[16] Marc Brown: *„Algorithm Animation“*. 1987. MIT Press, Cambridge.

[17] Nils Faltin: „Aktives *Lernen von Algorithmen mit interaktiven Visualisierungen*“. pp. 121 - 137 in: K. Mehlhorn and G. Snelting (Ed.): „Informatik 2000 - Jahrestagung der Gesellschaft für Informatik“. 2000. Springer.

[18] Nils Faltin: „Learning Algorithms with Algorithm Simulations“. 2001. http://www-cg-hci-e.informatik.uni-oldenburg.de/~faltin/SALA/int_vis_alg_e.html

[19] Nils Faltin: „*Collection of Courseware for Computer Science (OLLI)*“. 2001. http://olli.informatik.uni-oldenburg.de.

A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations

Christopher Hundhausen[1] and Sarah Douglas[2]

[1] Laboratory for Interactive Learning Technologies,
Department of Information and Computer Sciences,
University of Hawai'I, Honolulu, HI 96822
hundhaus@hawaii.edu

[2] Human-Computer Interaction Laboratory,
Department of Computer and Information Science,
University of Oregon, Eugene, OR 97403–1202
douglas@cs.uoregon.edu

Abstract.

Computer science educators have traditionally used algorithm visualization (AV) software to create graphical representations of algorithms that are later used as visual aids in lectures, or as the basis for interactive labs. Typically, such visualizations are high fidelity in the sense that (a) they depict the target algorithm for arbitrary input, and (b) they tend to have the polished look of textbook figures. In contrast, low fidelity visualizations illustrate the target algorithm for a few, carefully chosen input data sets, and tend to have a sketched, unpolished appearance. Drawing on the findings of ethnographic studies we conducted in a junior-level algorithms course, we motivate the use of low fidelity AV technology as the basis for an alternative learning paradigm in which students construct and present their own visualizations. To explore the design space of low fidelity AV technology, we present a prototype language and system derived from empirical studies in which students constructed and presented visualizations made out of simple art supplies. Our prototype language and system pioneer a novel technique for programming visualizations based on spatial relations, and a novel presentation interface that supports reverse execution and dynamic mark-up and modification

1 Introduction

Algorithm visualization (AV) software supports the construction and interactive exploration of visual representations of computer algorithms, e.g., [1-5]. Traditionally, computer science instructors have used the software to construct visualizations that are later used either as visual aids in lectures, e.g., [6], or as the basis for interactive labs, e.g., [3]. More recently, computer science educators have advocated using AV software as the basis for "visualization assignments," in which students construct their own visualizations of the algorithms under study [5].

Inspired by social constructivist learning theory [7], we have explored a teaching approach that takes "visualization assignments" one step further by having students *present* their own visualizations to their instructor and peers for feedback and discussion [8]. To better understand this approach, we conducted a series of ethnographic studies in a junior-level algorithms course that included such assignments. Our findings have led us not only to endorse this teaching approach as an effective way of getting students involved in and excited about algorithms, but also to advocate a fundamental redesign of traditional AV software so that it better supports this teaching approach.

In this paper, we use the findings of our ethnographic studies to motivate the need for a new breed of AV technology that supports the construction and presentation of *low fidelity* visualizations, which are capable of illustrating a target algorithm for a few, carefully chosen input data sets, and which have an unpolished, sketched appearance. Drawing on the findings of our ethnographic studies and prior empirical studies, we then derive a set of specific requirements for low fidelity AV technology.

To demonstrate what low fidelity AV technology might look like, we next present SALSA, an interpreted, high-level language for constructing low fidelity AVs, along with ALVIS, an interactive, direct manipulation environment for programming in SALSA. In manifesting the design implications of our empirical research, SALSA and ALVIS pioneer a novel *spatial* approach to algorithm visualization construction, as well as a novel visualization presentation interface that supports reverse execution, and dynamic mark-up and modification. We conclude by presenting a taxonomy that places SALSA and ALVIS into the context of past work, and by describing some directions for future research.

2 Motivation

The redesign of AV software advocated in this paper was motivated by a pair of ethnographic field studies we conducted in consecutive offerings of a junior-level algorithms course at the University of Oregon ([8], ch. 4). For an assignment in these courses, students were required to construct their own visualizations of one of the divide-and-conquer, greedy, dynamic programming, or graph algorithms they had studied, and then to present their visualizations to their classmates and instructor during specially-scheduled presentation sessions. To study the use of AV technology in these courses, we employed a variety of ethnographic field techniques, including participant observation, semi-structured interviews, videotape analysis, and artifact analysis.

2.1 Ethnographic Study I: High Fidelity

In the first of our ethnographic studies, students used the *Samba* algorithm animation package [5] to construct *high fidelity* visualizations that (a) were capable of illustrating the algorithm for arbitrary input, and (b) tended to have the polished appearance and precision of textbook figures, owing to the fact that they were generated as a byproduct of algorithm execution. To construct such visualizations with Samba, students began by implementing their target algorithms in C++. Next, they annotated the algorithms with procedure calls that, when invoked, generated and updated their visualizations using Samba routines. Finally, they engaged in an iterative process of refining and debugging their visualizations. This process involved compiling and executing their algorithms, noting any problems in the resulting visualization, and modifying their C++ code to fix the problems.

In this first study, three key findings are noteworthy. First, students spent over 33 hours on average constructing and refining a single visualization. They spent most of that time steeped in *low-level graphics programming*—for example, laying out graphics objects in terms of Cartesian coordinates, and writing general-purpose graphics routines. Second, in students' subsequent presentations, their visualizations tended to stimulate discussions about *implementation details*—for example, how a particular aspect of a visualization was implemented. Third, in response to questions and feedback from the audience, students often wanted to back up and re-present parts of their visualizations, or to dynamically mark-up and modify them. However, conventional AV software like Samba is not designed to support interactive presentations in this way.

2.2 Ethnographic Study II: Low Fidelity

These observations led us to change the visualization assignments significantly for the subsequent offering of the course. In particular, students were required to use simple art supplies (e.g., pens, paper, scissors, transparencies) to construct and present *low fidelity* visualizations that (a) illustrated the target algorithm for just a few input data sets, and (b) tended to have an unpolished, sketched appearance, owing to the fact that they were generated by hand. (In prior work [9,10], we have called such low fidelity visualizations *storyboards*.)

In this second study, three key findings stand out. First, students spent only about six hours on average constructing and refining a single visualization storyboard. For most of that time, students focused on understanding the target algorithm's procedural behavior, and how they might best communicate it through a visualization. Second, rather than stimulating discussions about implementation details, their storyboards tended to mediate discussions about the *underlying algorithm*, and about how the visualizations might bring out its behavior more clearly. Third, students could readily go back and re-present sections of their visualizations, as well as mark-up and dynamically modify them, in response to audience questions and feedback. As a result, presentations tended to engage the audience more actively in interactive discussions.

2.3 Discussion: From High to Low Fidelity

Our study findings furnish three compelling reasons why constructing and presenting low fidelity visualizations constitutes a more productive learning experience in an undergraduate algorithms course than constructing and presenting high fidelity visualizations. First, low fidelity visualizations not only take far less time to construct, but also keep students more focused on topics relevant to an undergraduate algorithms course. Second, in student presentations, low fidelity visualizations stimulate more relevant discussions that focus on algorithms, rather than on implementation details. Finally, low fidelity visualizations are superior in interactive presentations, since they can be backed up and replayed at any point, and even marked-up and modified on the spot, thus enabling presenters to respond more dynamically to their audience.

3 Deriving Design Requirements

The findings reported in the previous section motivate the development of a new breed of AV technology that supports the construction and presentation of low fidelity visualizations. Pertinent observations from our ethnographic studies, as well as from our prior detailed studies of how students construct algorithm visualizations out of simple art supplies [9, 10], provide a solid empirical foundation for design.

In our ethnographic fieldwork, we gathered 40 low fidelity storyboards that were constructed using conventional art supplies, including transparencies, pens, and paper. The following generalizations can be made regarding their content:

- The storyboards consisted of groups of movable, objects of arbitrary shape (but most often boxes, circles, and lines) containing sketched graphics; regions of the storyboard, and locations in the storyboard, were also significant.
- Objects in storyboards were frequently arranged according to one of three general layout disciplines: *grids*, *trees*, and *graphs*.
- The most common kind of animation was simple movement from one point to another; pointing (with fingers) and highlighting (circling or changing color) were also common. Occasionally, multiple objects were animated concurrently.

These observations motivate our first design requirement:

R1:	*Users must be able to create, systematically lay out, and animate simple objects containing sketched graphics.*

With respect to the process of visualization construction, we made the following observations in our previous studies of how humans construct “low fidelity” visualizations out of art supplies [9, 10]:

- Storyboard designers create objects by simply cutting them out and/or sketching them, and placing them on the page.
- Storyboard designers never size, place or move objects according to Cartesian coordinates. Rather, objects are invariably sized and placed *relative* to other objects that have already been placed in a storyboard.

These observations motivate our second and third design requirements:

R2:	*Users must be able to construct storyboard objects by cutting and sketching; they must be able to position objects by direct placement.*

R3:	*Users must be able to create storyboards using spatial relations, not Cartesian coordinates.*

Finally, with respect to the process of visualization execution and presentation, observations made both in our ethnographic studies, and in our prior empirical studies [9, 10], suggest the following:

Rather than referring to program source code or pseudocode, storyboard presenters tend to simulate their storyboards by paying close attention to, and hence maintaining, important *spatial relations* among storyboard objects. For example, rather than maintaining a numeric looping variable, storyboard designers might stop looping when an arrow advances to the right of a row of boxes.

- Storyboard presenters provide verbal play-by-play narration as they run their storyboards. In the process, they frequently point to storyboard objects, and occasionally mark-up their storyboards with a pen.
- The audience often interrupts presentations with comments or questions. To clarify their comments and questions, they, too, point to and mark up the storyboard.
- In response to audience comments and questions, presenters pause their storyboards, or even fast-forward or rewind them to other points of interest.
- Audience suggestions often lead to on-the-spot modifications of the storyboard—for example, changing a color scheme, adding a label, or altering a set of input data.

These observations motivate the following two requirements, which round out our list:

R4:	*The system must support an execution model based on spatial, rather than algorithmic logic.*

R5:	*The system must enable users to present their storyboards interactively. This entails an ability to execute storyboards in both directions; to rewind and fast forward storyboards to points of interest; and to point to, mark-up, and modify storyboards as they are being presented.*

4 Prototype Language and System

The requirements just outlined circumscribe the design space for a new breed of *low fidelity* AV technology. To explore that design space, we have developed a prototype language and system for creating and presenting low fidelity algorithm visualizations.

The foundation of our prototype is SALSA (Spatial Algorithmic Language for StoryboArding), a high-level, interpreted language for programming low fidelity storyboards. Whereas conventional *high fidelity* AV technology requires one to program a visualization by specifying explicit mappings between an underlying "driver" program and the visualization, SALSA enables one to specify *low fidelity* visualizations that drive themselves; the notion of a "driver" algorithm is jettisoned altogether. In order to support visualizations that drive themselves, SALSA enables the layout and logic of a visualization to be specified in terms of its *spatiality*—that is, in terms of the spatial relations (e.g., *above*, *right-of*, *in*) among objects in the visualization.

The SALSA language is compact, containing just three data types and 12 commands. SALSA's three data types model the core elements of the art supply storyboards observed in the empirical studies discussed in the previous sections:

1. *Cutout*. This is a computer version of a construction paper cutout. It can be thought of as a movable scrap of construction paper of arbitrary shape on which graphics are sketched.
2. *Position*. This data type represents an x,y location within a storyboard.
3. *S-struct*. A *spatial structure* (*s-struct* for short) is a closed spatial region in which a set of cutouts can be systematically arranged according to a particular spatial layout pattern. The prototype implementation of SALSA supports just one s-struct: *grids*.

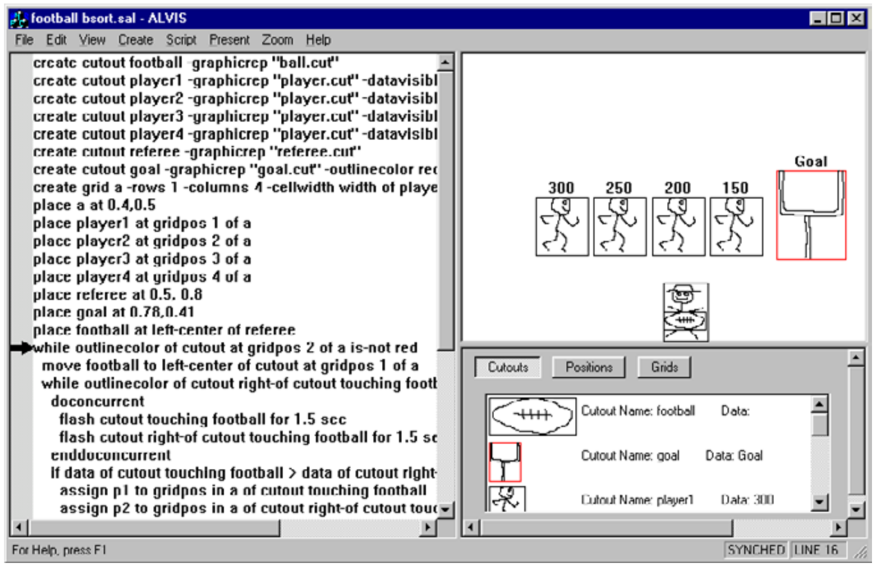
SALSA's command set includes *create*, *place*, and *delete* commands for creating, placing and deleting storyboard elements; an *if-then-else* statement for conditional execution; *while* and *for-each* statements for iteration; and *move*, *flash*, *resize*, and *do-concurrent* statements for animating cutouts.

The second key component of the prototype is ALVIS (<u>AL</u>gorithm <u>VI</u>sualization <u>S</u>toryboarder), an interactive, direct manipulation front-end interface for programming in SALSA. Figure 1 presents a snapshot of the ALVIS environment, which consists of three main regions:

1. *Script View* (left). This view displays the SALSA script presently being explored; the arrow on the left-hand side denotes the line at which the script is presently halted—the "insertion point" for editing.
2. *Storyboard View* (upper right). This view displays the storyboard generated by the currently-displayed script. The *Storyboard View* is always synchronized with the *Script View*. In other words, it always reflects the execution of the SALSA script up to the current insertion point marked by the arrow.
3. *Created Objects Palette* (lower right). This view contains an icon representing each cutout, position, and grid that has been created thus far.

The ALVIS environment strives to make constructing a SALSA storyboard as easy as constructing a homemade storyboard out of simple art supplies. To do so, its conceptual model is firmly rooted in the physical metaphor of "art supply" storyboard construction. An important component of this metaphor is the concept of *cutouts* (or *patches*; see [11]): scraps of virtual construction paper that may be cut out and drawn on, just like real construction paper. In ALVIS, users create storyboards by using a graphics editor (Figure 2a) to cut out and sketch cutouts, which they lay out in the *Storyboard View* by direct manipulation. They then specify, either by direct manipulation or by directly typing in SALSA commands, how the cutouts are to be animated over time.

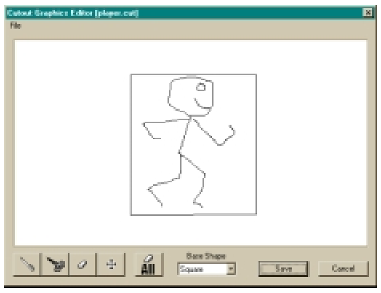
Likewise, ALVIS strives to make presenting a SALSA storyboard to an audience as easy and flexible as presenting an "art supply" storyboard. To that end, ALVIS's



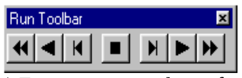
```
create cutout football -graphicrep "ball.cut"
create cutout player1 -graphicrep "player.cut" -datavisibl
create cutout player2 -graphicrep "player.cut" -datavisibl
create cutout player3 -graphicrep "player.cut" -datavisibl
create cutout player4 -graphicrep "player.cut" -datavisibl
create cutout referee -graphicrep "referee.cut"
create cutout goal -graphicrep "goal.cut" -outlinecolor rec
create grid a -rows 1 -columns 4 -cellwidth width of playe
place a at 0.4,0.5
place player1 at gridpos 1 of a
place player2 at gridpos 2 of a
place player3 at gridpos 3 of a
place player4 at gridpos 4 of a
place referee at 0.5, 0.8
place goal at 0.78,0.41
place football at left-center of referee
while outlinecolor of cutout at gridpos 2 of a is-not red
 move football to left-center of cutout at gridpos 1 of a
 while outlinecolor of cutout right-of cutout touching foott
  doconcurrent
   flash cutout touching football for 1.5 sec
   flash cutout right-of cutout touching football for 1.5 sc
  enddoconcurrent
  If data of cutout touching football > data of cutout right-
   assign p1 to gridpos in a of cutout touching football
   assign p2 to gridpos in a of cutout right-of cutout touc
```

football bsort.sal - ALVIS
File Edit View Create Script Present Zoom Help
300
250
200
150
Goal
Cutouts
Positions
Grids
Cutout Name: football Data:
Cutout Name: goal Data: Goal
Cutout Name: player1 Data: 300
For Help, press F1
SYNCHED
LINE 16

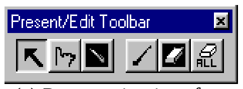
Fig. 1. A Snapshot of a session with ALVIS



(a) Cutout graphics editor



(b) Execution control interface



(c) Presentation interface

Fig. 2. Elements of the ALVIS interactive environment

presentation interface supports four features that are taken for granted in "art supply" presentations, but that are notably absent in conventional AV technology. First, using ALVIS's execution interface (Figure 2b), a presenter may reverse the direction of storyboard execution in response to audience questions and comments. Second, ALVIS provides a conspicuous "presentation pointer" (fourth tool from left in Figure 2c) with which the presenter and audience members may point to objects in the storyboard as it is executing. Third, ALVIS includes a "mark up pen" (third tool from left in Figure 2c) with which the presenter and audience members may dynamically annotate the storyboard as it is executing. Finally, presenters and audience members may dynamically modify a storyboard as it is executing by simply inserting SALSA commands at the current insertion point in the script.

5 Example Use of the Language and System

Perhaps the best way to provide a feel for the features and functionality of SALSA and ALVIS is through an illustrative example. In this section, we use SALSA and ALVIS to create and present the “football” storyboard (see Fig. 3) of the bubble sort algorithm we observed in prior empirical studies [9,10].

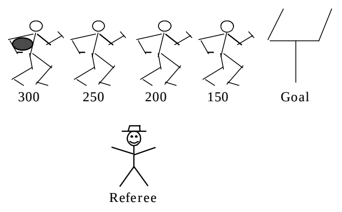


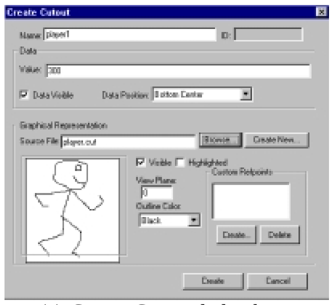
Fig. 3. The Football Bubblesort storyboard

The “football” storyboard models the bubble sort algorithm in terms of a game of American football. Elements to be sorted are represented as football players whose varying weights (written below each player) represent element magnitudes. At the beginning of the game, the players are lined up in a row. The referee then tosses the ball to the left-most player in the line, who becomes the ball carrier. The object of the game is to “score” by advancing the ball to the end of the line. If the ball carrier is heavier than the player next in line, then the ball carrier simply tackles the next player in line, thereby switching places with him. If, on the other hand, the ball carrier is lighter than the player next in line, then the ball carrier is stopped in his tracks, fumbling the ball to the next player in line. This process of ball advancement continues until the ball reaches the end of the unsorted line of players. Having found his rightful place in the line of players, that last player with the ball tosses the ball back to the referee. A pass of the algorithm’s outer loop thus completes. If, at this point, there are still players out of order, the referee tosses the ball to the first player in line, and another pass of the algorithm’s outer loop begins.

5.1 Creating the Storyboard Elements

We begin by creating the cutouts that appear in Figure 2: four players, a football, a referee, and a goal post. With respect to the football players, our strategy is to create one “prototype” player, and then to clone that player to create the other three players. To create the prototype player, we select *Cutout…* from the *Create* menu, which brings up the *Create Cutout* dialog box (see Figure 4a). We name the cutout “player1,” and use the *Cutout Graphics Editor* (Figure 2a) to create its graphics in the file “player.cut”: a square scrap of construction paper with a football player stick figure sketched on it. In addition, we set the data attribute to “300,” and decide to accept all other default attributes. When the *Create Cutout* dialog box is dismissed, ALVIS inserts the following SALSA create statement into the *SALSA Script View*:

```
create cutout player1 –graphic-rep “player.cut” –data “300”
```



Name: player1 ID:

Data

Value: 300

[✓] Data Visible Data Position: Bottom Center

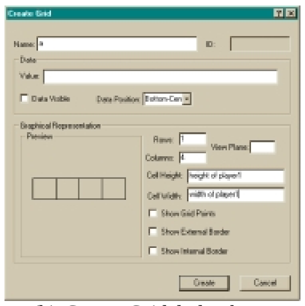
Source File: player.cut

[✓] Visible [ ] Highlighted

View Plane: 0

Outline Color: Black

(a) *Create Cutout* dialog box



Name: a ID:

Data

Value:

[ ] Data Visible Data Position: Bottom-Cen

Rows: 1 View Plane:

Columns: 4

Cell Height: height of player1

Cell Width: width of player1

[ ] Show Grid Points

[ ] Show External Border

[ ] Show Internal Border

(b) *Create Grid* dialog box

Fig. 4. Dialog boxes for creating SALSA objects

In addition, the `player1` cutout appears in the *Created Objects Palette*, and the execution arrow (in the *SALSA Script View*) is advanced to the next line, indicating that the create statement has been executed.

We now proceed to clone `player1` three times. For each cloning, we first select the `player1` icon in the *Created Objects Palette*, and then choose *Clone…* from the *Create* menu. This causes the *Create Cutout* dialog box to appear, with all attribute settings matching those of `player1`. In each case, we change the name (to "player2", "player3", and "player4", respectively), and the data (to "250", "200", and "150", respectively), while leaving all other attributes alone. After dismissing the *Create Cutout* dialog box each time, a new `create` statement appears in the script—for example,

```
create cutout player2 as clone of player1 –data "250"
```

In addition, the new cutout appears in the *Created Objects Palette*, and the execution arrow advances to the next line. We proceed to create the football, referee and goal post in the same way.

5.2 Placing the Storyboard Elements

The next step is to place the cutouts in the storyboard. In order to lay the players out in a row, we use a *grid* s-struct, which we create by choosing *Grid…*from the *Create* menu and then filling in the *Create Grid* dialog box (see Figure 4b). We name the grid `a`, and give it one row and four columns, with a cell height and width corresponding to the height and width of `player1`. We accept all other default attributes, and click on the "Create" button to create the grid.

We now place all of the objects we have created into the storyboard. First, we drag and drop the grid to an acceptable location in the middle of the storyboard; the corresponding `place` statement appears in the script, with the execution arrow advancing to the next line. With the grid in place, it is straightforward to position the players at the grid points:

```
place player1 at position 1 of a
place player2 at position 2 of a
place player3 at position 3 of a
place player4 at position 4 of a
```

Finally, we drag and drop the referee to a reasonable place below the row of football players; we drag and drop the football to the left center position of the referee; and we drag and drop the goal to a position to the right of the line of football players. Figure 1a shows the ALVIS environment after all `place` statements have been executed.

5.3 Programming the Spatial Logic

We are now set to do the SALSA programming necessary to animate the storyboard. ALVIS users may program much of this code through a combination of dialog box fill-in and direct manipulation. However, in the presentation that follows, we will focus on the SALSA code itself in order to demonstrate the expressiveness of the language.

As each player reaches his rightful place in the line, we want to turn his outline color to green. Since we have set the outline color of the goal post (the right-most cutout in the storyboard) to green, we know that when the outline color of the cutout immediately to the right of the ball carrier is green, the ball carrier has reached the end of the line, and we are done with a pass of bubble sort's inner loop. In SALSA, we may formulate this as a `while` loop:

```
while outlinecolor of cutout right-of cutout touching
   football is-not green
     --do body of inner loop of bubblesort (see below)
endwhile
```

We know that we are done with all passes of the outer loop when all but the first player in line has a green outline color. In SALSA, we may express this logic as another `while` loop:

```
while outlinecolor of cutout at position 2 of a is-not green
      --do body of outer loop of bubblesort (see below)
endwhile
```

Now we come to the trickiest part of the script: the logic of the inner loop. We want to successively compare the ball carrier to the player to his immediate right. If these two players are out of order, we want to swap them, with the ball carrier maintaining the ball; otherwise, we want the ball carrier to fumble the ball to the player to his immediate right.

The following SALSA code makes the comparison, either fumbling or swapping, depending on the outcome:

```
if data of cutout touching football > data of cutout right-of
                      cutout touching football --swap players
  assign p1 to position in a of cutout touching football
  assign p2 to position in a of cutout right-of cutout
    touching football
  doconcurrent
    move cutout touching football to p2
    move cutout right-of cutout touching football to p1
    move football right cellwidth of a
  enddoconcurrent
else --fumble ball to next player in line
  move football right cellwidth of a
endif
```

All that remains is to piece together the outer loop. At the beginning of the outer loop, we want to toss the ball to the first player in line. We then want to proceed with the inner loop, at the end of which we first set the outline of the player with the ball to green (signifying that he has reached his rightful place), and then toss the ball back to the referee. Thus, the outer loop appears as follows:

```
move football to left-center of cutout at position 1 of a
--inner while loop (see above) goes here
set outlinecolor of cutout touching football to green
move football to top-center of referee
```

5.4 Presenting the Storyboard

Using ALVIS's presentation interface, we may now present our "football" storyboard to an audience for feedback and discussion. Since nothing interesting occurs in the script until after the *Storyboard View* is populated, we elect to execute the script to the first line past the last `place` command by positioning the cursor on that line and choosing "Run to Cursor" from the *Present* menu. The first time through the algorithm's outer loop, we walk through the script slowly, using the "Presentation Pointer" tool to point at the storyboard as we explain the algorithm's logic. Before the two players are swapped, we use the "Markup Pen" tool to circle the two elements that are about to be swapped.

Now suppose an audience member wonders whether it might be useful to flash the players as they are being compared. Thanks to ALVIS's flexible animation interface and dynamic modification abilities, we are able to test out this idea on the spot. As we attempt to edit code, which lies within the script's inner `while` loop, ALVIS recognizes that a change at this point will necessitate a re-parsing of that entire loop. As a result, ALVIS backs up the script to the point just before the loop begins. At that point, we insert the following four lines of code:

```
doconcurrent --flash players to be compared
  flash cutout touching football for 1 sec
  flash cutout right-of cutout touching football for 1 sec
enddoconcurrent
```

We can now proceed with our presentation without having to recompile the script, and without even having to start the script over from the beginning.

6 Related Work

Beginning with Brown's BALSA system [1], a legacy of interactive AV systems have been developed to help teach and learn algorithms, e.g., [2-5]. SALSA and ALVIS differ from these systems in two fundamental ways.

First, the visualization construction technique pioneered by ALVIS and SALSA differs from those supported by existing systems. To place ALVIS and SALSA into perspective, Figure 5 presents a taxonomy of AV construction techniques. These may be divided into three main categories:

1. *Algorithmic*. Algorithmic construction involves the specification of mappings between an underlying (implemented) algorithm, and a visualization. Most existing AV systems support one of the four algorithmic techniques (see [12] for a review): *predefined*, *annotative declarative* or *manipulative.*

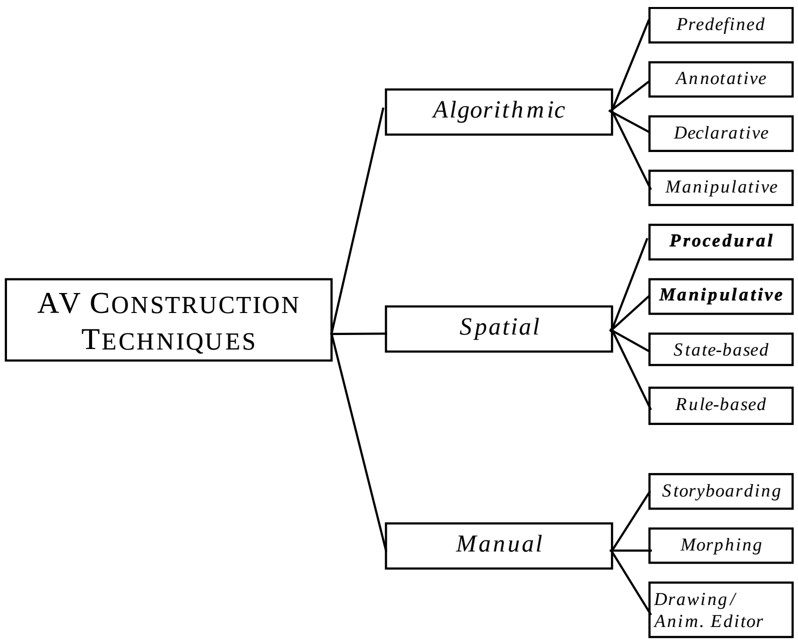


Fig. 5. A taxonomy of AV construction techniques

2. *Spatial*. In stark contrast to algorithmic techniques, spatial techniques completely abandon the use of an underlying "driver" algorithm that generates a visualization as a byproduct of its execution. Instead, spatial techniques specify a visualization in terms of the *spatiality* of the visualization itself. The SALSA language pioneers a *procedural* approach to spatial construction, while ALVIS supports a *manipulative* approach by enabling many components of SALSA programs to be programmed by direct-manipulation. In a similar vein, Michail's Opsis [13] supports a *state-based* visual programming approach in which binary tree algorithms are constructed by manipulating of abstract visual representations, while Brown and Vander Zanden [14] describe a *rule-based* approach in which users construct visualizations with graphical rewrite rules.
3. *Manual*. In *manual* AV construction, one abandons a formal underlying execution model altogether. Thus, there exists no chance of constructing AVs the work for general input; AV execution is entirely under human control. Manual construction techniques include *storyboarding* [9] on which ALVIS and SALSA are based; *morphing* [15], and *drawing and animation editors*—most notably, Brown and Vander Zanden's [14] specialized drawing editor with built-in semantics for data structure diagrams.

The second key difference between SALSA and ALVIS and existing AV technology lies in their support for visualization presentation. Almost without exception, existing AV technology has adopted Brown's [1] animation playback interface, which allows one to start and pause an animation, step through the animation, and adjust animation speed. As we have seen, in order to support interactive presentations, ALVIS's animation control interface goes well beyond this interface by supporting reverse execution, and dynamic markup and modification. Notably, in most existing AV systems, modifying an animation entails changing and recompiling source code, which is seldom feasible within the scope of an interactive presentation.

7 Summary and Future Work

In this paper, we have used the findings of ethnographic studies of an undergraduate algorithms course to develop a key distinction between high and low fidelity algorithm visualizations, and to motivate a shift from high fidelity to low fidelity AV technology as the basis for assignments in which students construct and present their own visualizations. Based on empirical studies of how students construct and present low fidelity visualizations made from simple art supplies, we have derived a set of requirements for low fidelity AV technology. To explore the design space circumscribed by these requirements, we have presented SALSA and ALVIS, a prototype language and system for constructing and presenting low fidelity visualizations. SALSA and ALVIS introduce a novel spatial technique for visualization creation, as well as a novel interface for presenting visualizations to an audience.

Although they demonstrate the key features of low fidelity AV technology, SALSA and ALVIS are clearly works in progress. In future research, we would like to focus our efforts in three key areas. First, prior to implementing SALSA, we conducted paper-based studies of SALSA's design with undergraduate algorithms students. In future research, we need to subject ALVIS to iterative usability testing in order to verify and improve the usability of its design. Second, we originally designed ALVIS with the intention of using it with a large electronic whiteboard (e.g., the "SmartBoard," http://www.smarttech.com), so that groups of students and instructors could engage in collaborative algorithm design. In such a setting, ALVIS would need to be used with some sort of stylus (pen) as its input device. In future research, we would like to explore the potential for ALVIS to be used in this way. Finally, SALSA and ALVIS are presently frail research prototypes; they were not implemented to be robust systems suitable for use in the real world. Thus, an important direction for future research is to improve both the completeness and the robustness SALSA and ALVIS, so that they may ultimately be used as the basis for assignments in an actual algorithms course. See http://lilt.ics.hawaii.edu/lilt/software/alvis/index.html for up-to-date information on our progress.

Acknowledgments. This research was conducted as part of the first author's dissertation [8], supervised by the second author. The first author is grateful for the second author's inspiration and astute guidance on the project. SALSA and ALVIS were programmed in Microsoft® Visual C++™ using the Microsoft® Foundation Classes. We used the Cygwin port of Gnu Flex and Gnu Bison to generate the SALSA interpreter. Thanks to Hank Bennett for programming a significant portion of ALVIS, and to Ted Kirkpatrick for providing invaluable help with debugging ALVIS.

References

1. M. H. Brown, *Algorithm animation*. Cambridge, MA: The MIT Press, 1988.
2. J. Stasko, “Tango: A framework and system for algorithm animation.” Ph.D. Dissertation (Tech. Rep. No. CS-89-30), Department of Computer Science, Brown University, Providence, RI, 1989.
3. T. Naps, “Algorithm visualization in computer science laboratories,” in *Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education*. New York: ACM Press, 1990, pp. 105-110.
4. G. C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun, “Pavane: A system for declarative visualization of concurrent computations,” *Journal of visual languages and computing*, vol. 3, pp. 161–193, 1992.
5. J. T. Stasko, “Using student-built animations as learning aids,” in *Proceedings of the ACM Technical Symposium on Computer Science Education*. New York: ACM Press, 1997, pp. 25-29.
6. M. H. Brown and R. Sedgewick, “Progress report: Brown University Instructional Computing Laboratory,” in *ACM SIGCSE Bulletin*, vol. 16, pp. 91-101, 1984.
7. J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*. New York: Cambridge U. Press, 1991.
8. C. D. Hundhausen, “Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course.” Ph.D. Dissertation (Tech. Rep. No. CIS-99-07), Department of Computer and Info. Science, University of Oregon, Eugene, 1999. Available at http://lilt.ics.hawaii.edu/~hundhaus/dis/.
9. S. A. Douglas, C. D. Hundhausen, and D. McKeown, “Toward empirically-based software visualization languages,” in *Proceedings of the 11th IEEE Symposium on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 342-349.
10. S. A. Douglas, C. D. Hundhausen, and D. McKeown, “Exploring human visualization of computer algorithms,” in *Proceedings 1996 Graphics Interface Conference*. Toronto, CA: Canadian Graphics Society, 1996, pp. 9-16.
11. M. van de Kant, S. Wilson, M. Bekker, H. Johnson, and P. Johnson, “PatchWork: A software tool for early design,” in *Human Factors in Computing Systems: CHI 98 Summary*. New York: ACM Press, 1998, pp. 221-222.
12. G. C. Roman and K. C. Cox, “A taxonomy of program visualization systems,” *IEEE Computer*, vol. 26, pp. 11-24, 1993.
13. A. Michail, “Teaching binary tree algorithms through visual programming,” in *Proceedings of the 12th IEEE Symposium on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 38-45.
14. D. R. Brown and B. Vander Zanden, “The Whiteboard environment: An electronic sketchpad for data structure design and algorithm description,” in *Proceedings of the 1998 IEEE Symposium on Visual Languages*. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 288-295.
15. W. Citrin and J. Gurka, “A low-overhead technique for dynamic blackboarding using morphing technology,” *Computers and Education*, pp. 189-196, 1996.

Towards a Taxonomy of Network Protocol Visualization Tools

Pilu Crescenzi and Gaia Innocenti

Dipartimento di Sistemi e Informatica,
Università degli Studi di Firenze,
Via C. Lombroso 6/17,
50134 Firenze, Italy
{piluc, innocent}@dsi.unifi.it

Abstract.

Due to the rising importance of visualization for teaching purposes, several network protocol visualization applications and applets are flowing on the Internet. So far, no taxonomy has been developed to classify this rather broad material. In this paper, we propose a taxonomy of network protocol visualization resources, to be used either by software visualization researchers to classify their tools or by educators to easily determine if a particular visualization actually satisfies their teaching needs. The taxonomy (which has been obtained by analyzing stand-alone applications and applets available on the Internet) is mainly intended to integrate previously existing taxonomies of algorithm visualization tools by mostly considering some characteristic features which are specific of the considered subject of network protocols.

1 Introduction

Many scientific studies have pointed out that algorithm understanding can be significantly improved if simulations and animations of various kinds assist the mental work [9,5,7]. In this context, network protocol visualization tools, as a particular kind of algorithm visualization tools, should be used as a support to basic course materials in teaching courses on computer networks, which usually cover the protocols and the software architecture layers in a rather theoretic way. For this reason, several network protocol visualization applications and applets are flowing on the Internet. However, everyone is writing tools for his/her own needs and there is no effort to create a sort of repository that would help people finding the best suitable animation for their needs. Such a repository would be extremely useful for teaching purposes, since the educators would have a tool by which they can choose, among different visualizations of the same protocol, the one best suiting their needs.

As a first step towards creating a repository of network protocol visualization resources, in this paper we are going to define a tentative taxonomy that should help software visualization researchers to classify their tools and educators to easily determine if a particular visualization actually satisfies their teaching needs.

The proposed taxonomy (which has been obtained by analyzing stand-alone applications and applets available on the Internet) is mainly intended to integrate one of the most known taxonomy of software visualization tools, which was described by Price, Small and Baecker [12]. Our new taxonomy intends to point out the characteristic features which are specific of the considered subject of network protocols.

2 Key Concepts

The term *protocol* can be used in several contexts, and it usually refers to the entity which defines the format and the order of messages exchanged between two or more communicating elements, as well as the actions taken on the transmission and/or receipt of a message or other event.

In particular, a *computer network protocol* provides a communication service that higher-level objects (such as application processes) use to exchange messages. Each protocol usually defines two different interfaces: the *service interface* to the other objects that want to use its services (for example, the interface of the Transmission Control Protocol, in short TCP, which is used by the File Transfer Protocol, in short FTP) and the *peer interface* which defines the form and meaning of *messages* exchanged between protocol peers to implement the communication service.

Messages are what application level protocols exchange in accomplishing their tasks: messages can contain data or may perform a control function. Messages can be broken into smaller sized units, called *packets* (further decomposable in smaller aggregates called *frames*) which traverse communication links and packet switches (also known as *routers*) to reach the destination.

A *network topology* specifies the set of computer elements (hosts and routers) belonging to the network and the set of links they are connected by, which provide the means to transport messages. A network topology can be visualized as a graph: for this reason we will often refer to nodes meaning hosts or routers.

Each link can then be characterized by its *bandwidth*, which is the number of bits that can be transmitted over the link in a certain period of time, and by its *latency* (or *delay*), which corresponds to how long it takes a single bit to propagate from one end of the connection to the other.

The *network state* (or network configuration) is characterized by the actual values of the variables of each process involved in the protocol under execution and by the identity of the messages in transit along the connections.

We have recalled here the basic concept underlying the computer network subject, which is sufficient to understand the main features that a network protocol visualization tool should deal with. Anyway, we refer the interested reader to one of the several textbooks currently available (see, for example, [8,11,13]).

3 Characteristic Features

In general terms, a network protocol visualization tool is a software that simulates the behavior of a protocol running over a network by means of a graphical representation. A network protocol visualization tool is somehow different from a network visualization tool: the latter is mainly concerned with the graphical representation of the topology of a network (see, for example, Otter [6]), while the former is developed to assure (or it tries to assure) a good visualization of how protocols behave on a network.

Network protocol visualizers can be seen as a special kind of algorithm visualization tools which focus over networks, messages and connections. The specific subject they illustrate obviously influences their characteristic features: as a consequence, network protocol visualizers should be analyzed according to a new taxonomy, which would extend existing taxonomies of software visualization tools.

In this paper we start from one of the most complete taxonomies of software visualization tools, which was introduced by Price, Small and Baecker [12], and we describe a possible extension of this taxonomy that could be used to categorize network protocol visualizers. This extension is intended to describe in a more detailed and suitable way the characteristic features of a network protocol visualization resource.

In the next sections we present the new features that could be specifically observed in a network protocol visualizer and we propose how to integrate them in the original taxonomy. We then show how the extended taxonomy should be applied in the description of various network protocol visualization resources, including applications and applets.

4 The Extended Taxonomy

In the following sections we present the criteria that we propose to use in order to define our network protocol visualizers' taxonomy.

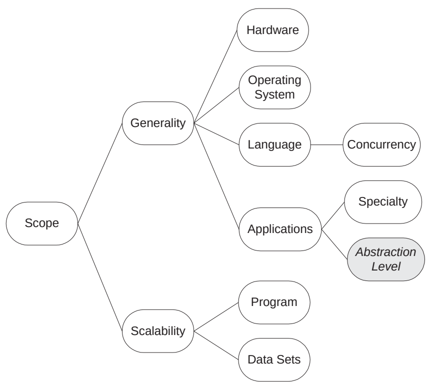
The referred taxonomy described in [12] classifies software visualization systems according to six main top level categories, each of which can be further organized in subcategories in a hierarchical structure. This organization lets the taxonomy be expandable, in order to permit new criteria to be added to fit the description of visualization systems of specific areas. Thanks to this opportunity, we can naturally add the peculiarities of network protocol visualization tools without redesigning the entire tree.

The six top level mentioned categories are:

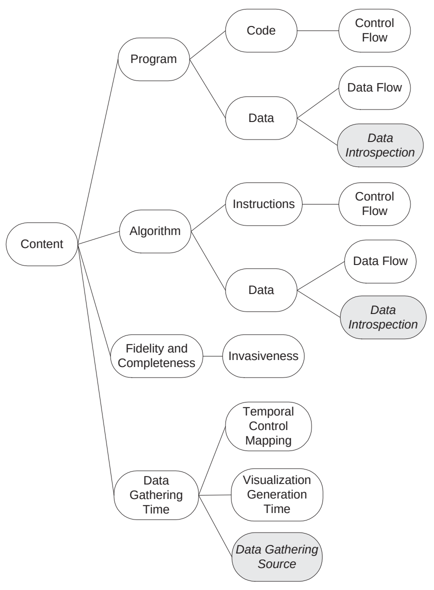
– Scope.
– Content.
– Form.
– Method.
– Interaction.
– Effectiveness.

The *Scope* criterion specifies the range of programs that the visualization system takes as input for the visualization itself while the *Content* criterion describes what subset of information about the software is visualized by the system that we are analyzing. The *Form* category illustrates in detail what the characteristics of the output of the system are. The *Method* criterion describes the style in which the visualizer specifies the visualization and the way in which the actual visualization and the code are connected. The taxonomy illustrates the *Interaction* of the user with the system and it describes its *Effectiveness*, that is, how well the visualizer communicates information to the user.

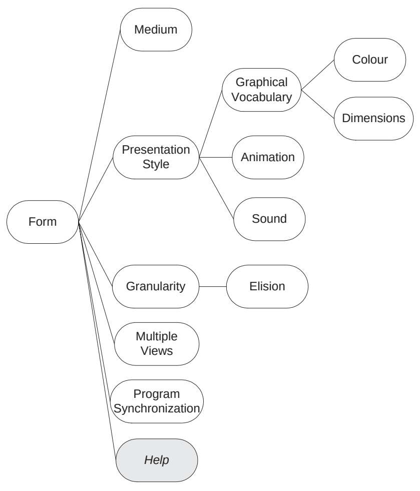
According to this subdivision, we will insert our new criteria in order to extend the taxonomy to fit the peculiar features of network protocol visualization tools: In the next subsections we are then going to describe each of these new criteria and to specify their positions in within the above described taxonomy.



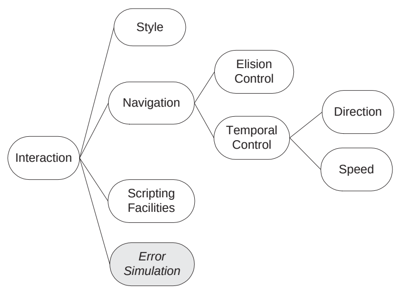
The added Abstraction Level criterion.



The added Data Gathering Source and Data Introspection criteria.



The added Help criterion.



The added Error Simulation criterion.

Fig. 1. The extended taxonomy.

4.1 Abstraction Level

A computer network architecture can be defined by means of the seven layers described in the OSI standard [15]: a network protocol implements a functionality assigned to a given layer. At each layer, the kind of data that are exchanged differs according to contents and meaning: the physical layer, for example, handles the transmission of raw bits, while the data link layer groups bits in a larger set called frame.

A visualizer can illustrate network protocols related to a particular layer. Hence, the visualizations can be classified as **highly abstract** (if they deal with the application layer data) or **lowly abstract** (if they deal with the physical layer data). This information is conveyed in the *Abstraction Level* property (see Fig. 1). In the original taxonomy, the sub-category *Applications* of the Scope criterion defined the restrictions on the kinds of user programs that can be visualized, and its further sub-category *Specialty* specified the kinds of programs the system is particularly good at visualizing. In our case, the Specialty feature will indicate network protocols in general, or the particular protocol being visualized (if the system being analyzed is addressed to a specific network protocol). The new sub-category Abstraction Level of the Applications sub-category will further indicate the layer level that the visualization tool can visualize.

4.2 Data Introspection

Network protocol visualization tools should be applied to network topologies of every reasonable dimension (starting from networks with only two nodes). As the topology increases its number of components, the visualization should accordingly become more simplified in order to let it be still comprehensible: hence, each node and each link should be visualized in the most compact way.

To handle big topologies, a network protocol visualization tool should give users a level of introspection to browse the contents of messages, nodes and links.

To this aim, we extend the top level category Content by adding a new sub-category (see Fig. 1) which we call *Data Introspection* and which describes the basic network elements that the network protocol visualization tool can show. Since the level of introspection can be specified either for program visualizations or for algorithm visualizations (in our case, we should better say network protocol visualizations), we duplicate the new sub-category to let it extend the Data feature of both kinds of visualization.

Messages. Most of the times, messages flowing in the network channels are represented in rectangle shapes to simplify the global network view, since at any time a lot of different messages can be present on the same link. Then, users can have a closer view of the contents of each message actually circulating in the network by means of additional information. The further information can be displayed in pop-up windows appearing when users click on the message itself, or it can be visualized in special areas of the main window.

Nodes. The same argument can be repeated for network nodes. Due to the big number of properties that each node may have (i.e. name, address, state, and so on), most of the visualizers follows the policy of showing them on demand.

Links. Another component that can hide information in order to simplify the protocol visualization is the connection link. A visualizer should let users investigate the contents of each edge of the network. Commonly, the additional information related to a link is its identity, its bandwidth and its latency.

4.3 Data Gathering Source

A network protocol tool, as a specialized visualization tool, can be classified according to the data it gathers. The taxonomy [12] considers the time in which data are gathered, i.e. at compile time, at run time or both. Network protocol visualizers, besides, need a more detailed criterion, based on the source from which data can be retrieved, that should be considered if data are gathered at run time. Thus, our extended taxonomy inserts a new sub-category (see Fig. 1) to the element Data Gathering Time of the top level category Content, that we call *Data Gathering Source*. As a matter of fact, the specialized visualization tools we are interested in can be fed by two main sources: *real networks or files*. The first feature is a peculiarity of tools such as network protocol visualizers, which can actually be employed as simulators of real network traffic, that is generated according to some protocol. Indeed, input files can be specified for every kind of visualization tool, but they were not taken into account in the taxonomy we are extending. Input files can be defined as trace files or as protocol files. A trace file generally indicates an input file that is coded in a special format, which can be understood by the visualization system and which represents the trace of the protocol execution. Instead, a protocol file represent the implementation of the protocol to be visualized.

4.4 Help

Despite the user friendliness of the graphical interface of a visualization tool, its first use is almost never immediate. Therefore, a characteristic feature that should be considered in the definition of a taxonomy is the level of supported help that a visualization system provide to users. Then, we propose to insert a new sub-category (that we will call *Help*) in the top level category Form (see Fig. 1). This new element should be used to identify two distinguishing features:

- **Tool help**: Some tools have documentation on their usage and on the functionalities they provide. If the visualization system is used as a supporting aid by teachers, the classification should also cover the possible integration with course materials, that could be useful to evaluate its effective use. Besides, this category should indicate if the tool is still supported by its authors, in order to be confident of a further help. In particular, in classifying a network protocol visualizer, the help documentation should describe also how to load and to execute a network protocol visualization, how to interact with it, how to choose, if possible, the protocol to be visualized, and so on.
- **Protocol help**: Since a visualization tool can be used to display many different network protocols, some authors provide the tool with descriptions on how to run a specific protocol, on the node and link properties that can be visualized (such as the number of messages to be sent, the number of messages to be received, the level of priority) and their meanings. Sometimes the visualizer's help supports even the protocol description. Due to the nature of network protocols, two main kinds of help description can be visualized:
  - **Natural language description:** This is the kind of protocol description often presented in textbooks by means of a natural language (i.e. English).

- **Pseudo-code description:** This is a more formal way of describing a protocol and it is less commonly used.

4.5 Error Simulation

A characteristic feature of network protocol visualizations is the ability to deal with errors. Since a visualization tool should actually describe how a protocol works and since messages flowing in a network are subject to transmission errors or noise, a good visualization tool should consider this situation. To this aim, we propose to extend the taxonomy [12] including the sub-category *Error Simulation* to the top level category Interaction (see Fig. 1).

The Error Simulation feature can be implemented in two different ways:

– **Sending an error message:** In this way users can trigger off an error message whenever they like, just selecting the node that will generate and send the error message.
– **Setting an error probability:** In this way users can set the error probability associated to a channel (corrupting all messages circulating on that particular link) or to a node (corrupting all messages sent by that node). Users have no power in determining when the error will be issued (unless they set the probability to 1): In this case, the visualization gets very close to a real situation.

5 Network Protocol Visualization Tools

To illustrate our extended taxonomy we will refer to several network protocol visualization tools, either general systems (applications) or specific systems (applets). The choice of the analyzed resources has been driven mainly by the broad use of the tools and by their educational application. We would have liked to study and report more tools, but some of them were commercial products or they had restrictions on the supported platforms, so that they could not be tested.

In Tables 1 to 7 we describe how we can classify each of the analyzed tools according to the categories of our extended taxonomy. In the description of the reported values, for each network protocol visualization tool we discuss the added criteria in more depth, in order to best explain their actual application. For a more detailed description of the original categories, we refer to the taxonomy [12].

5.1 NAM

The first tool we refer to is NAM (Network ANimator) [4]. NAM is a Tcl/TK based animation tool for viewing network simulation traces and real world packet traces. NAM began at the Lawrence Berkeley National Laboratory (LBNL) in Berkeley in 1990 and it has evolved substantially over the past few years. The NAM development effort was an ongoing collaboration with the VINT (Virtual InterNetwork Testbed) project [3]; currently, it is being developed at ISI (Information Sciences Institute), University of Southern California, as part of the SAMAN and Conser projects.

Abstraction Level. The abstraction level of NAM is strictly related to the particular protocol that users want to visualize. NAM supports the visualization of protocols related to many layers defined in the OSI standard, such as the physical, data link, transport and application layers. The stop and wait algorithm, for example, displays the sending and the receiving of packets: in this case, NAM is used to visualize the data layer.

Table 1. The Scope category.

	Generality							**Scalability**	
		Hardware	**Operating System**	**Language**		**Applications**		**Program**	**Data Sets**
					Concurrency	**Specialty**	***Abstraction Level***		
NAM	System	PC, Sun	FreeBSD, Linux, Windows, SunOS, Solaris	TclTk and C++	No	Network protocols	High/Low depending on the protocol	Experienced large program	Experienced large data sets
APJ	System	Mac, PC, Sun	supporting JVM	JAVA and APJ library	No	Network protocols	High/Low	Experienced educational programs	Experienced up to 20 nodes
CNET	System	SUN, PC	Unix, Linux	TclTk and C	No	Network protocols	High/Low depending on the protocol	For educational purposes	For educational purposes
Univ. Mumbai	Applet	supporting JVM	supporting JVM	Java	No	CSMA CD protocol	Low	Fixed	Up to 15 nodes
SIT-IIT	Applet	supporting JVM	supporting JVM	Java	No	CSMA CD protocol	Low	Fixed	Up to 4 nodes
Univ. Berne	Applet	supporting JVM	supporting JVM	Java	No	CSMA CD protocol	Low	Fixed	Fixed - 3 nodes
Polyt. Warsaw	Applet	supporting JVM	supporting JVM	Java	No	CSMA CD protocol	Low	Fixed	Fixed - 3 nodes

Data Introspection. NAM provides a good level of data introspection, since it allows users to see additional information just by clicking on each network component. Clicking on a link generates a pop-up frame which shows the link identity, bandwidth and delay. From here, users can choose between viewing a bandwidth utilization graph or viewing a link loss graph of one directional link of the bidirectional link. Moreover, clicking on a message produces a pop-up window which displays the size of the message, the message identity and the time it was sent. Finally, the only additional information visualized when clicking on a node is the node identity. In despite of this, it is possible to monitor a protocol agent (that is, a node), a packet, or a connection link to observe all available object-specific information. Monitors are associated to a particular component until they are removed by the users or until the related object is destroyed. Setting a monitor on a data packet labels the packet so that users can easily watch its flow through the network and the queues.

Table 2. The Content category - part 1.

	Program/Algorithm					
		Code/Instructions		**Data**		
			Control Flow		**Data Flow**	***Data Introspection***
NAM	Algorithm	Annotations	Annotations and exchanged messages/packets	Textual and graphical	Messages/packets exchanged	Links, messages, nodes
APJ	Both	Variables/Pseudo-code	Highlighting pseudocode/Exchanged messages	Textual and graphical	Messages/packets exchanged	Links, nodes
CNET	Algorithm	Annotations and statistics	Annotations and statistics and exchanged messages	Textual and graphical	Messages/packets exchanged and statistics	Links, nodes
Univ. Mumbai	Algorithm	No	No	Textual and graphical	Frames sent and textual	No
SIT-IIT	Algorithm	No	No	Textual and graphical	Frames sent and textual	No
Univ. Berne	Algorithm	No	No	Graphical	Frames sent	No
Polyt. Warsaw	Algorithm	No	No	Textual and graphical	Frames sent and textual	No

Data Gathering Source. NAM visualizes trace files, which contain topology information, such as node, link and packet traces. NAM was developed to visualize trace files generated by the NS simulator [14], another tool produced by the VINT project. Indeed, since the format of a trace file is of public domain, everyone can produce (automatically or by hand) a correct trace file to feed NAM. Besides, NAM can visualize trace files produced by the emulation functionality of NS, which can be introduced into a real live network to get traffic information.

Table 3. The Content category - part 2.

	Fidelity and Completeness		Data Gathering Time			
		Invasiveness		Temporal Control Mapping	Visualization Generation Time	*Data Gathering Source*
NAM	High	No	Run-time	Dynamic-dynamic	Post-mortem	Trace file
APJ	High	No	Run-time	Static-static	Live	Protocol file
CNET	High	No	Run-time	Dynamic-dynamic	Live	Protocol file
Univ. Mumbai	High	No	Run-time	Dynamic-dynamic	Live	User input
SIT-IIT	High	No	Run-time	Dynamic-dynamic	Live	User input
Univ. Berne	High	No	Run-time	Dynamic-dynamic	Live	User input
Polyt. Warsaw	High	No	Run-time	Dynamic-dynamic	Live	User input

Help. NAM provides a tool help: From the visualizer's main window menu bar, users can select the help item and then they can view the help related to the use of NAM's functionalities, together with the version of NAM that they are using and a short history of the tool, including the actual maintainers of the system. In spite of this, NAM does not provide any kind of description of the protocol it is visualizing, neither a natural language description nor a pseudo-code description. The only help visualized is the list of events that are happening during the protocol execution, such as the starting of the protocol, the messages being sent and the ending of the protocol.

Table 4. The Form category.

	Medium	Presentation Style					Granularity		Multiple Views	Program Synchr.	*Help*
		Graphical Vocabulary			Animation	Sound		Elision			
			Colour	Dimensions							
NAM	Workstation	Line, circle, box	Yes	2D	Yes	No	Zoom, node, messages, link	Yes	Yes	Yes	Tool help. No protocol help.
APJ	Workstation	Line, box	Yes	2D	No	No	Node, link	Yes	Yes	No	Protocol help.
CNET	Workstation	Image, line	Yes	2D	Yes	No	Node, link	Yes	Yes	No	Some tool documentation
Univ. Mumbai	Workstation	Line, box	Yes	2D	Yes	No	No	No	No	No	Tool help, protocol doc., applet design
SIT-IIT	Workstation	Line, box	Yes	2D	Yes	No	No	No	No	No	Tool help, protocol doc.
Univ. Berne	Workstation	Line, box	Yes	2D	Yes	No	No	No	No	No	No
Polyt. Warsaw	Workstation	Line, box	Yes	2D	Yes	No	No	No	No	No	Few tool help

Error Simulation. Unfortunately, NAM simulates the occurring of message losses only in case the simulation trace file, given as input to the visualization tool, simulates the presence of errors. NAM itself does not let users set any error probability nor explicitly generate an error message.

Table 5. The Method category.

	Visualization				**Connection**		
	Specification Style	**Intelligence**	**Tailorability**		**Technique**	**Code Ingnorance Allowance**	**System Code Coupling**
				Customization Language			
NAM	Trace file	Low	Zooming, editing	Interactive	Read file	Low: produce a trace file	Low
APJ	Hand coding and use of library	Low	Editing	Interactive	Invasive	Low: use APJ library	High
CNET	Hand coding and use of library	Medium	Links' and nodes' attributes	Interactive	Invasive	Low	High
Univ. Mumbai	Hand coding	Low	Editing	Interactive	Invasive	High	High
SIT-IIT	Hand coding	Low	Editing	Interactive	Invasive	High	High
Univ. Berne	Hand coding	Low	Set frame size	Interactive	Invasive	High	High
Polyt. Warsaw	Hand coding	Low	Set parameter values	Interactive	Invasive	High	High

Table 6. The Interaction category.

	Style	**Navigation**				**Scripting Facilities**	***Error Simulation***
			Elision Control	**Temporal Control**			
				Direction	**Speed**		
NAM	Menus, hot areas, buttons	Yes	Yes	Yes	Yes (length of a step)	Save layout	No
APJ	Menus	Yes	Yes	Yes	Yes	Save layout	Yes
CNET	Hot areas, buttons	Yes	Yes	No	Yes	Add annotations, save topology	Yes
Univ. Mumbai	Buttons	No	No	No	No	No	No
SIT-IIT	Buttons	No	No	No	No	No	No
Univ. Berne	Graphical	No	No	No	No	No	No
Polyt. Warsaw	Buttons	No	No	No	No	No	No

5.2 APJ

The second network protocol visualizer we refer to is APJ, which has been developed in Java at the University of Florence [2]. APJ allows the user to visualize pre-compiled network protocols, which have to be written using a library of functions provided with the tool. APJ has been used as a teaching aid in a network class in the Computer Science program at the University of Florence.

Abstraction Level. APJ's abstraction level depends on the particular protocol being visualized. APJ provides a library to deal with messages. The abstraction level should be categorized as high level. In spite of that, messages are only displayed in textual mode and they have information just about their content. As a consequence, they could be used to describe a low layer protocol.

Table 7. The Effectiveness category.

	Purpose	Appropriateness & Clarity	Empirical Evaluation	Production Use
NAM	Education + Research	Subjective	Some	Since 1990
APJ	Education	Subjective	Not found	Since 1999
CNET	Undergraduate Education	Subjective	Not found	Since 1991
Univ. Mumbai	Final course project	Few	Not found	1998-99
SIT-IIT	Edu. project	Subjective	Not found	Jan. 2001
Univ. Berne	Education	Few	Not found	1998
Polyt. Warsaw	Education	Subjective	Not found	Not found

Data Introspection. APJ is a network protocol visualization tool which allow users to have a more detailed view of the links and nodes displayed in the main window. Each network node can be further inspected by means of a new window that displays the values of variables used for its implementation, the state of its input channels and the pseudo-code of its last executed action. To have a more precise view of the contents of a connection link, the user should control the input channel panels contained in the window of the two nodes connected by the desired link.

Data Gathering Source. APJ retrieves the information to be visualized by loading a file. The identity of the file can be of two kinds: it can be the Java compiled file implementing the network protocol that the user wants to visualize, or it can be a file in a special format. The second kind of file specifies the Java class file containing the protocol implementation and it specifies some additional information about the protocol visualization (i.e. error probability, speed, pseudo-code file).

Help. APJ provides help at the protocol level. If the programmer has previously specified the pseudo-code which describes the protocol being visualized, the tool can show in textual mode the actions taken by the nodes of the network during the execution of the protocol. The lower part of the main window (and the optional window associated to each node) is dedicated to this purpose.

Error Simulation. APJ provides a good level of error simulation. To this aim, the tool allows the user to set the error probability associated to each channel of the network, that is, the probability according to which a message flowing in a link can be transformed into an error message. To set the error probability, users should select a menu item, thus getting a dialog window which allows them to choose a connection link and to set its error probability. The error probability associated to a link can be modified at any time during the execution of the protocol and the occurrence of an error is visualized by coloring in red the channel containing the message that has been converted to an error.

5.3 CNET

The third network protocol visualization tool which we are going to analyze is CNET [10]: a network simulator which enables experimentation with data link, network (routing), transport and session layer protocols. CNET has been developed at the

University of Western Australia since 1991 and it is nowadays maintained by its author. CNET has been used as a supporting tool in teaching network undergraduate classes in many institutions ever since.

Abstraction Level. CNET allows the user to visualize both physical and application layer protocols. To this aim the tool shows either messages or frames, which have been sent and received by each host.

Data Introspection. CNET allows the user to investigate the contents of nodes and links, just by clicking at the graphical elements of the network which are represented in the main window. The first time the user clicks on a node, CNET displays a new window which describes the node characteristics and which is organized in three panels. The upper part lets the user set three message properties (that are its rate, its minimum and its maximum size) and the central panel shows the number of messages and bytes which have been generated, correctly received and wrongly received and the corresponding bandwidths. The lower panel of the window displays a sequence of textual messages describing the actions taken by the node. A new window appears even if the user clicks on a link. Since links are bidirectional, clicking on an extreme of the link activates the window related to the link starting from the nearest node. Each window shows the number of frames and bytes which have been transmitted and received so far, their bandwidth and the number of errors that have been introduced. The lower panel of the window lets the user set some sort of weight: that is, the link's costs per byte and per frame and the probability of frame corruption and of frame loss. To homogeneously set the characteristic features of all the nodes and links of the network, the user can display the default node attributes and the default link attributes windows, which are identical to the ones just described. CNET does not provide facilities to access the identity of messages or frames circulating in the network, although it displays in the main window the percentage of messages delivered correctly.

Data Gathering Source. CNET visualizes network protocols that the user has to specify in the command line. CNET retrieves information about the protocol to show from a special format file, which specifies the topology of the network and the file containing the actual implementation of the protocol.

Help. CNET does not offer a good level of help. It does not describe neither a tool help nor the protocol being visualized. Although, the full distribution that can be downloaded from the CNET official web site [1] includes some documentation about the tool's functionalities and the library it provides.

Error Simulation. CNET provides a good level of error simulation. The user can set the probability that a frame on a link will be corrupted and the probability that it will be lost altogether. To set these values, the user can set every link at a time or he can set default values that will be common to all the links in the network.

5.4 Applets

In this section we are going to categorize, according to our new taxonomy, some applets that we found on the web and that we have collected in the repository of network protocol visualization tools that we are constructing
(see http://www.dsi.unifi.it/∼innocent/repository/). The analyzed applets have been grouped so that each group lists various implementations of the same network protocol. In this way we can have a global view of which features have been focussed by each applet and then we can decide which of the presented applets could better fit our requirements.

In the following, we will describe in detail four Java applets that implement the *CSMA CD* protocol. Furthermore, as previously done for the network protocol visualization systems we have described, we illustrate how the new criteria of our specialized taxonomy fit into the classification of each applet.

CSMA CD protocol. CSMA CD is an acronym for Carrier Sense Multiple Access with Collision Detect and it is the protocol used for carrier transmission access in Ethernet networks, in which a set of nodes send and receive frames over a shared link. The *carrier sense* means that all the nodes can distinguish between an idle and a busy link, and *collision detect* means that a node listens as it transmits and can therefore detect when a frame it is transmitting has interfered (i.e. collided) with a frame transmitted by another node. If a collision occurs, each device then waits a random amount of time and retries until it successfully gets its transmission sent.

University of Mumbai. This CSMA CD applet was realized by a team of three students of the Department of Computer Engineering, Thadomal Shahani Engineering College, University of Mumbai, India, in 1998-99. The applet was part of a report for a project entitled "Tools for Network and Protocol Simulation", submitted in partial requirement for the degree of Bachelor of Engineering (B.E.). The project team members were: Aditya Tuli, Arun Narayanan, Kaushal Vaidya and Vineet Verma.

- *Abstraction Level:* the protocol deals with frames of data, belonging to the data link layer of the OSI standard, which is one of the lowest layers. Thus, the abstraction level of the applet is low.
- *Data Introspection:* the applet does not provide any level of data introspection.
- *Data Gathering Source:* the user has to specify either the network topology (adding nodes), either the frames of data to be sent or the rate of data transmission. Thus, the applet retrieves data information directly from the user input.
- *Help:* the user can view the help on using the applet by clicking on the Help button. The authors provide also some documentation on the implemented protocol and on the applet design.
- *Error Simulation:* the applet does not allow to simulate the occurring of errors.

SIT - IIT. This CSMA CD applet was developed as part of the xNet Project at the School of Information Technology, Indian Institute of Technology, Bombay, India. The main goal of the xNet Project is to develop web-based tools for eXplaining Networking concepts and technologies, thereby enabling self-learning and distance education. The project is coordinated by Sridhar Iyer. The authors of this applet were the students: B. Anuradha, C. Manoj Kumar and T. Chitti Babu.

- *Abstraction Level:* due to the CSMA CD protocol, the abstraction level of the applet is low.
- *Data Introspection:* the applet does not provide any level of data introspection.
- *Data Gathering Source:* as in the previous applet, the user has to specify the network topology (adding nodes). Besides, the user may define the frame size to be used. Then, the applet retrieves data information from the user input.
- *Help:* the applet provides a very summarized tool help and some documentation on the CSMA CD protocol.
- *Error Simulation:* the applet does not consider the opportunity to simulate the occurring of errors.

University of Berne. This CSMA CD applet was developed as a student project for the computer network course held by Torsten Braun at the Institute of Computer Science and Applied Mathematics, University of Berne, Switzerland, in 1998. The authors of this applet were: L. Ammon and C. Neuhaus.

- *Abstraction Level:* due to the nature of the visualized protocol, the abstraction level of the applet is low.
- *Data Introspection:* the applet does not provide any level of data introspection.
- *Data Gathering Source:* the applet defines a fixed topology made of three nodes. The user has to click on each node to let it generate data frames. The user may set the size of the data frames.
- *Help:* the authors did not provide any kind of help documentation.
- *Error Simulation:* the applet does not provide any facility to simulate the occurring of errors.

Polytechnic of Warsaw. This CSMA CD applet was found as part of the course material of the Telecommunication course held by Krzysztof M. Brzezinski at the Institute of Telecommunications of the Polytechnic of Warsaw, Poland.

- *Abstraction Level:* as in the previous analyzed applets, the abstraction level of the CSMA CD visualization is low.
- *Data Introspection:* the applet does not provide any level of data introspection.
- *Data Gathering Source:* the applet defines a fixed topology made of three nodes. The user has to click on each node to let it start sending data frames. The user may set parameter values such as the bus length, the frame size and the transmission rate.
- *Help:* the author provided a very summarized help documentation.
- *Error Simulation:* the applet does not provide any facility to simulate the occurring of errors.

6 Conclusions and Future Research

In this paper we have proposed some criteria which can be used in order to create a taxonomy of network protocol visualization resources. The criteria are mainly intended to integrate previously existing taxonomies of algorithm visualization tools by mostly considering some characteristic features which are specific of the considered subject of network protocols.

The next step of our research will be to apply the proposed criteria in order to create a Web-based repository of network protocol visualization resources. For the moment, the repository is at an early stage and it contains only a little part of the available tools. This repository is intended to be used either by software visualization researchers to classify their tools or by educators to easily determine if a particular visualization actually satisfies their teaching needs.

References

1. The cnet network simulator. *http://www.cs.uwa.edu.au/cnet/*.
2. P. Crescenzi, G. Innocenti, and S. Pasqualetti. Implementing and Visualizing Network Protocols. In *Proc. of the First Program Visualization Workshop*, pages 193–206, 2001.
3. D. Estrin. *Virtual InterNetwork Testbed (VINT): methods and system*. ISI Proposal 96-ISI-05.
4. D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu. Network visualization with Nam, the VINT network animator. *IEEE Computer*, 33:63–68, 2000.
5. S. Hansen, D. Schrimpsher, and N.H. Narayanan. Learning algorithms by visualization: A novel apporach using animation-embedded hypermedia. In *Proc. of the 1998 International Conference of the Learning Sciences*, pages 125–130, Dec 1998.
6. B. Huffaker, E. Nemeth, and K. Claffy. *Otter: a general-purpose network visualization tool*. CAIDA: Cooperative Association for Internet Data Analysis, 1999.
7. C. Kehoe, J. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *Graphics,Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-99-10*, March 1999.
8. J. Kurose and K. Ross. *Computer Networking: A Top-down Approach Featuring the Internet*. Addison-Wesley, Reading, MA, 2000.
9. A.W. Lawrence, A.M. Badre, and J.T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proc. of the 1994 IEEE Symposium on Visual Languages*, pages 48–54, Oct 1994.
10. C.S. McDonald. A Network Specification Language and Execution Environment for Undergraduate Teaching. In *Proc. of the ACM Computer Science Education Technical Symposium '91, San Antonio, Texas*, pages 25–34, Mar 1991.
11. L.L. Peterson and B.S. Davie. *Computer Networks: A Systems Approach (2nd Ed.)*. Morgan Kaufmann Publishers, 1999.
12. B.A. Price, I.S. Small, and R.M. Baecker. Introduction to Software Visualization. In *Software Visualization: Programming as a multimedia experience*, pages 3–28. J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), MIT Press, 1997.
13. A.S. Tanenbaum. *Computer Networks (3rd Ed.)*. Prentice Hall, 1996.
14. The VINT Project. *The ns Manual (formerly ns Notes and Documentation)*. A collaboration between researchers at UC Berkley, LBL, USC/ISI, and Xerox PARC, March 2001.
15. H. Zimmerman. OSI reference model – The ISO model of architecture for open systems interconnection. *IEEE Trans. Comput.*, 28:425–432, 1980.

Understanding Algorithms by Means of Visualized Path Testing*

Ari Korhonen[1], Erkki Sutinen[2], and Jorma Tarhio[1]

[1] Department of Computer Science and Engineering
Helsinki University of Technology, Finland
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
{archie,tarhio}@cs.hut.fi

[2] Department of Computer Science
University of Joensuu, Finland
sutinen@cs.joensuu.fi

Abstract.

Visualization of an algorithm offers only a rough picture of operations. Explanations are crucial for deeper understanding, because they help the viewer to associate the visualization with the factual meaning of each detail. We present a framework based on path testing for associating instructive explanations and assignments with a constructive self-study visualization of an algorithm. The algorithm is divided into blocks, and a description is given for each block. The system contains a separate window for code, flowchart, animation, explanations, and control. Assignments are based on the flowchart and on the coverage conditions of path testing. Path testing is expected to lead into more accurate evaluation of learning outcomes because it supports systematic instruction in addition to more free trial-and-error heuristics. A qualitative analysis of preliminary experiences with the prototype indicates that the approach helps a student to reflect on her own reasoning about the algorithm. However, a prerequisite for an successful learning process with the environment is a motivating introduction, describing both the system and the main idea of the algorithm to be learned.

1 Introduction

A major challenge in Computer Science education is that of learning algorithmic thinking and a related skill, namely programming. While an algorithm is on a more abstract level than a program, they require a similar readiness from a student: she has to be able to comprehend the core, idea, or essence of a computational process, not just its step-by-step running-time behavior with a given input.

Quite often, a hindrance of learning is due to reasoning in a narrow or conventional way. A student concentrates on peripheral issues rather than strives

* The work was supported by the National Technology Agency, Finland.

her way into the nucleus of a new topic. In the case of programming, this might lead into an entirely incorrect misconception of a program's key idea.

Computer science educators have tried to solve the problem of learning an algorithm by the help of various visualization tools [4,6,8,11,17]. From a pedagogical perspective, one could divide them into two groups: the ones targeted for instruction and the ones for construction. The first category covers environments consisting of fancy ready-made animations with a possible teacher's voice explaining the algorithm. These animation packages fit well into a teacher-directed learning context. In the second category we have tools, which can be used by the learners themselves for designing their own visualizations; they work for more open-ended learner-centered settings.

In a self-study situation, a student makes use of both instructive and constructive learning tools. To compensate the lack of a teacher, a tool has to mimic a teacher by explaining an algorithm, if possible, on-the-fly. A crucial question is how to provide a student with an environment where she can freely study an algorithm and, at the same time, get instructive feedback on the essential stages of it.

The efficiency of the visual learning tools, whether instructive or constructive, is hard to assess. However, there seems to be a tentative resolution that at least constructive tools help a learner to get a deeper insight into a given algorithm or program. The problem is, however, that in most environments the effort of building an animation takes at least the same time as learning the algorithm in some other way.

Although a constructive animation environment may open the "black box" of a program, its abstract contents might still remain hidden for the visitor. Indeed, the learner will follow how the program runs on a given input, but that is usually not definitely the whole truth. As an example, consider an implementation of the bubble sort working on a sorted sequence.

In the process of understanding the essence of an algorithm or a piece of code, it would, therefore, be attractive to develop technologies which do not only take a learner as a visitor of a black box onto a single ride, corresponding to a program run with one input. Instead, the environment should welcome her as a season-card resident of the box, to get acquainted in the whole box, now as a "glass box". This means that the system should try to help the user to take all the typical paths of the program. This setting is closely related to path testing [2], a widely used software testing technique, where one considers various coverage measures based on paths in the control flow graph of a program.

Naturally, path testing could be used in many ways, and also in conjunction with an instructive or a constructive visualization tool. Basically, a semi-automatic guide could give hints on those paths which the student has not yet discovered. This mechanism could be used within a code editor as well as a visualization tool. Obviously, the bottom-up approach to learning an algorithm by different instances corresponding to different inputs raises questions. For example, what happens if the learner only tries out peripheral or trivial cases, even in a random way? To overcome these kinds of difficulties, a learning environment based on path testing should have instructive features. Especially, it should tutor the student to cover all the paths, and after that, following the tradition of Pólya [15], reason about the idea behind all these paths.

Different learners have various kinds of problems in understanding the character of a given program. A promising idea behind the path testing is that each learner can usually start with a path which is simple for her to understand, and then proceed to more demanding paths. Reasoning about the meaning of paths is probably an efficient way to transfer one's understanding above the current zone of proximal development [19]. This learning process can probably take place also within a community of learners, whether physical or virtual.

Intuitively, the idea of path testing comes from tracing potential paths in a flowchart representing a given program. This intuition might mislead one to suspect that the tested paths have to be presented visually on a flowchart. This is, however, not the case. In fact, the idea of path testing is independent of any single visual representation but rather reflects the abstract idea of an algorithm under study. Thus, it can be applied not only in various visualization tools, but virtually in any program concretization environment.

Unlike current technology trends for learning which aim at producing massive but mostly static learning materials for virtual universities, the idea of path testing is based on the need for tools which support learning of skills. Compared to learning knowledge from Web-stored digital libraries, path testing takes learners into an environment where they are supposed to learn and create new ideas by actively identifying novel paths.

The rest of the paper has been organized as follows. We start with introducing some basic concepts of path testing in Section 2. In Section 3 we sketch the guidelines of the learning environment of a new kind for studying algorithms. In Section 4 we describe a pilot environment we made to test our ideas in practice. Before discussing ideas on future development in Section 6, we report results of a qualitative evaluation of our prototype in Section 5.

2 Path Testing

Path testing [2,5] is a widely-used glass box testing technique, i.e. the source code of a program to be tested is known. A test case corresponds to a certain value combination of input variables. Each test case induces a path in the flow graph of the program. One considers the paths corresponding to a set of test cases under certain coverage conditions.

Test coverage analysis is the process of finding areas of a program not exercised by a test set, i.e. a set of test cases. This involves creating additional test cases to increase coverage and identifying redundant test cases that do not increase coverage. The aim is a complete coverage with a test set of minimal size.

There are several measures of test coverage. The most common ones are the following:

Statement coverage is satisfied, when every statement is executed at least once with the test set.

Branch coverage is satisfied, when every edge of the flow graph of the program is applied at least once with the test set.

Multiple condition coverage is satisfied, when every possible true/false combination of Boolean sub-expressions of each Boolean expression occurs.

Path coverage is satisfied, when the test set contains a test case for every possible control path in the flow graph of the program.

Clearly path coverage includes branch coverage which in turn satisfies statement coverage. Multiple condition coverage is similar to branch coverage but has better sensitivity to the control flow. However, multiple condition coverage does not necessarily imply branch coverage and path coverage does not guarantee multiple condition coverage in a general case.

Path coverage has the advantage of requiring very thorough testing. Because the number of paths is exponential to the number of branches, full path coverage is seldom useful. In practice weaker measures are used instead of the full path coverage.

As said above, the aim of coverage analysis is to find a test set of minimal size satisfying a certain coverage condition. Because of the difficulty of this optimization problem, one must be satisfied with approximate solutions in practice.

Example. Let us consider a code fragment of two lines with two input variables `a` and `b`:

```
if (a<4) or (b>5) then b:=b+1;
if b<2 then a:=a+7;
```

The following test sets of pairs (a, b) satisfy the coverage conditions presented above:

Statement coverage:	$\{(0, 0)\}$
Branch coverage:	$\{(0, 0), (4, 2)\}$
Multiple condition coverage:	$\{(0, 0), (3, 6), (4, 5), (4, 6)\}$
Path coverage:	$\{(0, 0), (0, 1), (4, 1), (4, 2)\}$

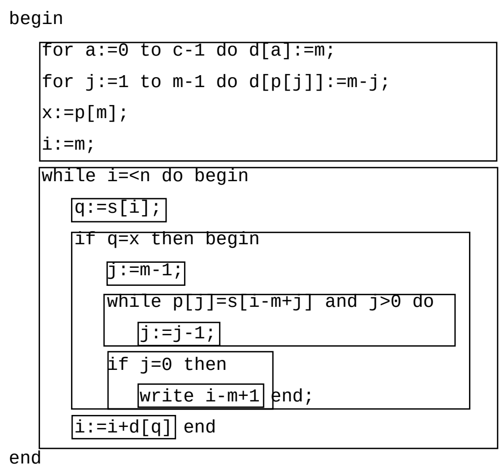
Note that in this case, multiple condition coverage does not hold for the test set satisfying path coverage and vice versa.

3 Guidelines of the Approach

Our aim is to present a new framework of self-study learning environments for algorithms. Our approach framework is partially based on path testing. As a prerequisite, the concepts of path testing should be presented in a self-explaining way so that the learner need not know anything about path testing before using an environment based on our framework.

The algorithm is divided into blocks. A basic block [1] is a sequence of statements in which flow of control enters at the beginning and leaves at the end without outgoing or incoming branching in the middle. In addition to the linear division to basic blocks, it is beneficial to consider a hierarchical division and apply stepwise refinement [18], because the hierarchy promotes learning of complex algorithms. On the lowest level a block is approximately a basic block. On the highest level there is only one block. Descriptive names could be associated with the blocks.

We use the Boyer-Moore-Horspool (BMH) algorithm [7] for string matching as an example algorithm in order to show how the idea of path testing supports the learning process. String algorithms provide Computer Science educators with a challenge. How to make students—even on an advanced level—understand the distinctive idea of an algorithm which consists only of few lines? A block hierarchy of BMH is given in Figure 1. Some of the blocks may be left without refinement, like the initialization on the first four lines in Figure 1.



```
begin
   for a:=0 to c-1 do d[a]:=m;
   for j:=1 to m-1 do d[p[j]]:=m-j;
   x:=p[m];
   i:=m;
   while i=<n do begin
      q:=s[i];
      if q=x then begin
         j:=m-1;
         while p[j]=s[i-m+j] and j>0 do
            j:=j-1;
         if j=0 then
            write i-m+1 end;
      i:=i+d[q] end
end
```

Fig. 1. Block hierarchy of the BMH algorithm.

An explanation describing pre and post conditions and the meaning of actions is given for the blocks. Besides the block descriptions, a general review of the algorithm and explanations of the meanings of the variables are also given. The environment contains a separate window for code, flowchart, animation, explanations, and control. The code window shows the code of the algorithm with the active line highlighted.

The flowchart window is a novel feature. The user may get several views of the execution. The first view shows a single path in the flowchart. Alternatively, the history of execution is shown as a colored worm creeping along the flow lines. The second view shows the activity of flow lines. Applied paths are shown such that the width of a line reflects the number of applications. Statement coverage can be checked with this view. The third view is for examining multiple condition coverage. Conditional branch nodes have a mark for every combination of Boolean primaries, which are typically relational expressions. The color of a mark indicates which combinations have occurred.

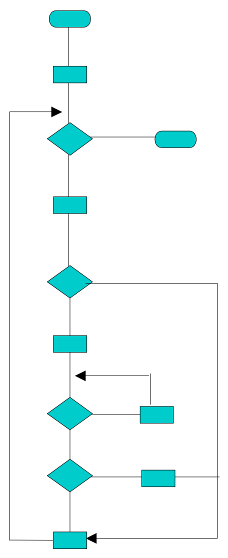


Fig. 2. Flow chart of the BMH algorithm.

The environment provides students with assignments in order to lead them to a more profound understanding and at same time to monitor their learning. Suitable assignments include:

– Give inputs for statement, branch, and multiple condition coverage.
– One of the blocks is left without explanation. Give an explanation.
– Explain how the algorithm proceeds in given situations.
– Some algorithm specific problems.

Because the assignments are prepared in advance, possible problems in the coverage of test sets can be ruled out. The answers consisting of test sets are fairly straightforward to check automatically.

There are several ways the environment can be used. In addition to giving traditional classroom presentation, one can define a homework assignment. In the case of the BMH algorithm, the student may be asked to define a pattern that satisfies statement coverage. In this example it turns out that statement coverage could be satisfied by only one pattern which occurs in the text. Generally this type of question could be rephrased by asking the student to define a set of patterns that satisfies the given condition, i.e., the statement, branch, multiple condition, or path coverage.

Also questions concerning some special features of the algorithm could be asked. In the case of the BMH algorithm, we could ask for a pattern that causes the algorithm "behave badly" (e.g. the algorithm examines each character of the text).

The assignments are formulated in such a way that in many cases a solution may be verified in the environment. After a student thinks that she has solved the given problem, she can simulate the algorithm in the environment, which provides immediate visual feedback of the correctness of the solution.

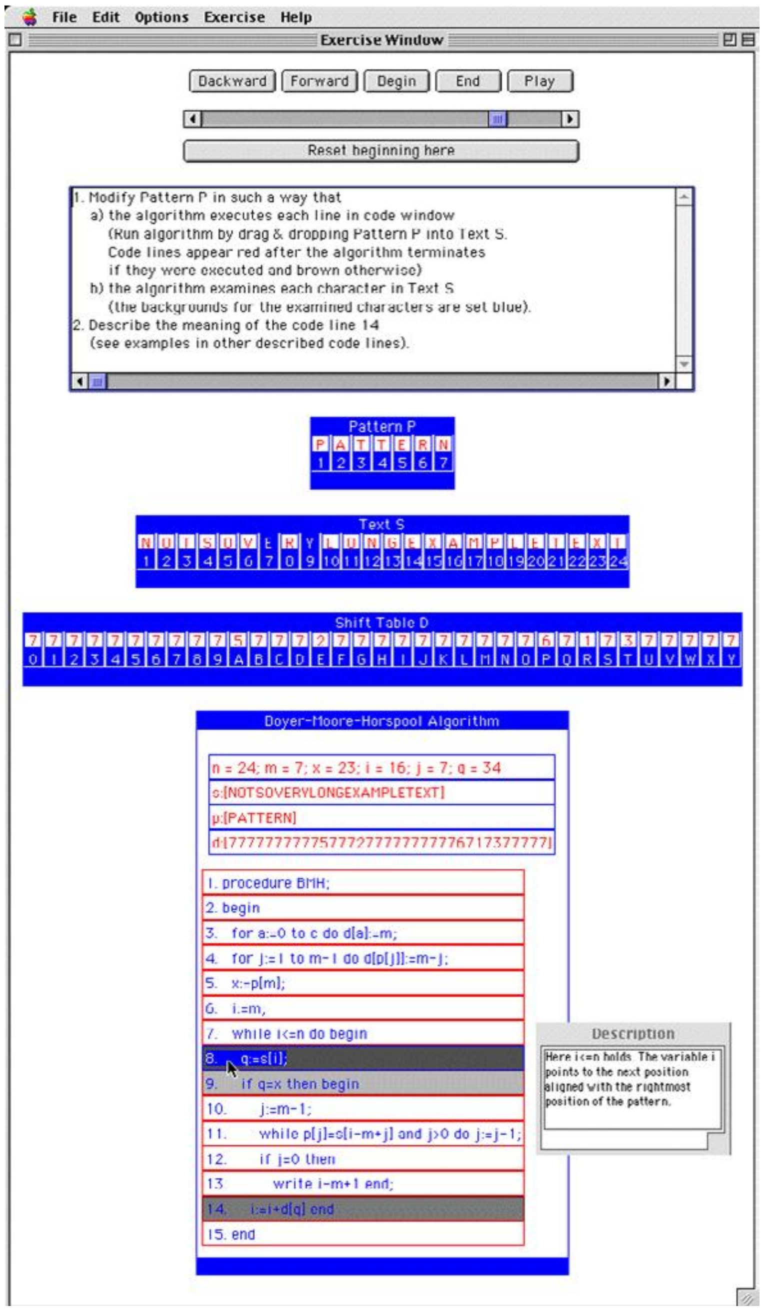
Questions dealing with explanations are open in the sense that there is not just a single correct solution for them. Therefore it is not possible to provide unique visual feedback about the correctness of a given answer. However, this kind of questions are suitable for classroom discussions.

4 Prototype

We implemented a prototype environment to test our ideas in practice. The prototype shows the current values of data structures in a discrete animation of the BMH algorithm. The system has separate code, control and explanation windows. The flowchart window was not implemented, but some of its functions were incorporated with the code window. Instead of a hierarchical block view of the algorithm, code lines are used as blocks.

We chose Matrix [10,12] for the application framework on top of which the prototype was implemented. Matrix is a platform independent visualization and simulation framework that provides extensive set of building blocks for visualizing data structures and algorithms. Even though Matrix is not primarily intended for code animation, but for concept animation and simulation of data structures, it turned out to be suitable for fast prototyping of the new system. The abstraction for visualizing string matching algorithms for this experiment took about 20 hours to complete. Additional 20 hours were used to improve the prototype based on the feedback gathered during testing. Thus, we expect that the creation of similar visualizations for another algorithm will take significantly less time, even if the author were not familiar with the framework. The abstraction for visualizing string matching algorithms has been included in the current version of the Matrix package which is freely available at Matrix homepage [14].

The visualization consists of six separate elements (see a screen snapshot in Figure 3) which are the animator components (buttons labeled *Backward, Forward, Begin, End, Play, Reset beginning here*, and the slider bar), the array *Pattern P*, the array *Shift Table D*, the two representations of the BMH algorithm (the array representation labeled *Text S* and the code view representation titled *Boyer-Moore-Horspool Algorithm*), and finally the optional text window labeled "Description". The user may freely alter the characters in arrays P and S. Finally, when ready to execute the algorithm, she just drag & drops the pattern array into the text array on the screen and the system invokes the



1. Modify Pattern P in such a way that
   a) the algorithm executes each line in code window
      (Run algorithm by drag & dropping Pattern P into Text S.
      Code lines appear red after the algorithm terminates
      if they were executed and brown otherwise)
   b) the algorithm examines each character in Text S
      (the backgrounds for the examined characters are set blue).
2. Describe the meaning of the code line 14
   (see examples in other described code lines).

Pattern P

P	A	T	T	E	R	N
1	2	3	4	5	6	7

Text S

N	O	T	S	O	V	E	R	Y	L	O	N	G	E	X	A	M	P	L	E	T	E	X	T
1	2	3	4	5	6	7	0	9	10	11	12	13	14	15	16	17	10	19	20	21	22	23	24

Shift Table D

7	7	7	7	7	7	7	7	7	7	5	7	7	7	2	7	7	7	7	7	7	7	7	7	7	6	7	1	7	3	7	7	7	7	7
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Doyer-Moore-Horspool Algorithm

```
n = 24; m = 7; x = 23; i = 16; j = 7; q = 34
s:[NOTSOVERYLONGEXAMPLETEXT]
p:[PATTERN]
d:[77777777775777277777777767173777777]
```

```
1. procedure BMH;
2. begin
3.   for a:=0 to c do d[a]:=m;
4.   for j:=1 to m-1 do d[p[j]]:=m-j;
5.   x:=p[m];
6.   i:=m;
7.   while i<=n do begin
8.     q:=s[i];
9.     if q=x then begin
10.      j:=m-1;
11.      while p[j]=s[i-m+j] and j>0 do j:=j-1;
12.      if j=0 then
13         write i-m+1 end;
14.    i:=i+d[q] end
15. end
```

Description

Here i<=n holds. The variable i points to the next position aligned with the rightmost position of the pattern.

Fig. 3. Screen snapshot of the prototype implementation.

actual method implementing the BMH algorithm with the given parameters. Now the visualization sequence appears in the animator and it could be examined through the animator interface. In addition, the execution of the algorithm could be repeated as many times as required with different inputs P and S.

4.1 Creating Code Visualization

The execution of the algorithm causes the animator to store all the method invocations during the execution in such a way that they could be revoked later on. Thus, the changes to the data structures involved could be restored. This and the issues dealing with data structure visualizations are taken care of the Matrix framework and is not discussed here any further. However, the code visualization is a new feature within Matrix, thus we examine that issue slightly closer.

The actual implementation of the target algorithm A should be written in Java as the Matrix is written in Java. However, the code visualization does not show this actual implementation to the user directly but an abstraction A' of it. This abstraction could be written in any language or it could be some kind of pseudo-code presentation or even a plain textual description. In order to make the connection between these two representations of the algorithm, the visualizer should annotate the actual algorithm A by "timestamping" the code lines of the abstraction A'. We chose this post-mortem technique [16] rather than live visualization because it allows the user to have complete control of the animation (speed, direction, levels of abstractions shown, etc.).

The abstract algorithm A' is defined by implementing the interface Code that serves as our abstraction for code visualization. Code includes a set of CodeLines that is another abstraction for a single code line representation. The actual algorithm should be annotated in such a way that the algorithm should invoke *CodeLine.stamp*(*int time*) method for each CodeLine just before the actual operation is performed. This assigns a new timestamp for the corresponding code line. Based on this information, Matrix is capable of visualizing the execution of the code as long as the implementation of Code can provide a clock (method *int getTime*()) that returns the "current time" at any point during the animation.

In addition, the CodeLine interface has a method that could return additional information about the state of the algorithm. Thus, the visualizer can provide a description about the state of the algorithm, the purpose of a single code line, questions concerning the algorithm, etc.

4.2 Examining of an Algorithm

To be able to examine the execution of an algorithm A stepwise, the algorithm is first run without visualization. At this point all the lines visited are shown with different color than those lines which were never executed. This was easy to accomplish because those lines never executed still have the timestamp zero and all the other lines have timestamp greater than zero. This approach demonstrates how the statement coverage is illustrated.

The six animator buttons and the slider bar allow the user to freely traverse through the animation sequence. The five buttons at top of the animator are self-explanatory. The sixth button “Reset beginning here” is used for adjusting the beginning of the animation sequence at given position and thus helps the user to, for example, mark and jump easily to the beginning of the last execution of the algorithm at any time. Moreover, the slider bar could be used to jump at any position in the animation sequence. This kind of interactive algorithm animation helps the user to figure out, for example, why some portions of the code were never executed.

As mentioned above, there are two representations for the BMH algorithm. One illustrates only the text S and the other that includes the algorithm. Because of the nature of Matrix, both of these conceptual views represent the very same physical object that is an instance of the class implementing the string matching abstraction. Thus, there exists only one implementation for the BMH algorithm even though we have two simultaneous representations. Moreover, while the execution of the algorithm makes changes to the shared physical data structures, both representations are updated synchronously. For example, in the upper array representation of the text the changes are expressed by darkening the array position i, if it is accessed at line 8 or 11 in the lower code representation.

In the code view the active line of execution is highlighted. Moreover, the lines executed just before the active line are highlighted with lighter shades to illustrate the progress of the program. In addition, an optional text box could be attached to each code line describing the state of the algorithm. In our demonstration this feature is used for describing the pre and post conditions and the meaning of actions for each code block as mentioned in Section 3.

5 Experiences

The prototype was tested with several CS major students, which did not have any prior knowledge about the topic. The first two students gave feedback that was used to improve the prototype before it was introduced to the third student. The final version was fine-tuned based on this latter test case and evaluated by some colleagues. This iterative and incremental design approach gave us the chance to validate and fine-tune the prototype enough to make all the critical design decisions. We discuss the final version shown in Figure 3 and also the improvements made, because the testing process showed that lack of some minor details could have dramatic influence on the learning curve.

The first and expected outcome was that it is not obvious that the students use the system as we expect them to do. For example, the first student tried to completely master the algorithm before starting to watch the visualized simulation. This was somewhat bizarre because we believed that it is the simulation that could be the philosopher’s stone to accomplish this task. However, the static code frame is a natural starting point, and the first student spent quite a long time simulating the algorithm in his mind, instead of using our system to do the simulation process. Finally, after the first student managed to solve the assignment, he was asked to teach the algorithm to the second one. Even though, the environment forced the first student to use the simulation to complete the assignment, he did not use the simulation capabilities for demonstration purposes while explaining the exercise to the second student. They even used pen and pencil while they discussed the algorithm. Again, the simulation came into the picture after the second student was sure enough of the correctness of his solution and he had the courage to try to solve the assignment.

However, the fact that we should encourage the students to freely explore the algorithm with the system was taken into account when the system was introduced to the third student. A few minutes were spent to illustrate the functionality before the student was let to try it on his own. This time the student started to use the visual debugger at once. He also managed to clarify for himself the logic of the algorithm just by running the algorithm once step by step with the default input. However, also this time the interaction between the user and the system increased during the session. The better he understood the algorithm the more he used the interactive capabilities of the system. The bottom line is that all of the students benefited most from the fact that the system could verify their thinking process and this way give feedback of their performance.

We also noticed that we must provide the student "the big picture" in order to get the learning process started. This is even more important in the case of interactive tools than in the case of traditional methods, because the user cannot predict the amount of material she has to go through. For the time being, our prototype does not include any overall description about the topic or the algorithm, thus the first student really had to struggle before he got some idea of how the algorithm works. Even though normally students are familiar with the topics to be studied, we believe that the overall idea should be provided in order to bring the context in students' mind.

Much of the observed experiences can be interpreted by referring to the individual differences in students' learning styles. Some prefer text-based information to visual one, others find holistic presentation a natural one compared to serial presentation. Basically, a learning environment should adapt to these differences. Adaptive mechanisms are usually rather hard to implement; however, simpler ideas solve the problem at least partly.

For example, texts and other representations can be redundant to some extent. This is because interactive tools, much like regular education, do not represent the whole content at once, but reveal it piece by piece. Thus, for example, the overall idea can be zoomed to highlight further details after the student has discerned the big picture. To quote a Latin proverb, Repetitio mater studiorum est.

We had an option to represent the actual algorithm in Java instead of an abstraction of it. However, representing only an abstraction allows more economic illustrations and hiding of unnecessary details. This leads to better understanding of the logic of an algorithm and reduce problems due to language-specific issues. Moreover, the algorithm may be presented in any language, even in a pseudo language. Of course, this approach requires additional work because of the mapping between the actual algorithm and the abstraction of it.

6 Future Perspectives

Since our path testing environment is at a stage of an early prototype, there are many alternatives for further development.

First of all, the path testing idea is not at all restricted to our current implementation. It could be incorporated, for example as a tutoring agent, into existing visualization tools. For example Jeliot [3,6] could be expanded to allow interruptions where a user could assign new values into variables. This way she could study what happens at a loop's end or similar steps of execution. This kind of on-the-fly path testing helps a student to concentrate on the part of the code which she is most interested in.

Path testing can also be used within other kinds of program concretization tools. This is particularly relevant in learning environments targeted at younger learners or K12 programs. For example a flow chart could be easily implemented as a three-dimensional playground where a student could adventure along pipelines. Also robotic learning environments for experimental programming, like Lego/Logo [9] or Empirica Control [13], could get new flavor when a user had to check all potential paths of a robot's behavior.

Since path testing clearly emphasizes the semantics of the program, it could also be applied to a visual proof of the program's correctness. This way the otherwise complicated concepts of semantics could be made more concrete and learned along with introductory programming.

The idea of path testing gives new opportunities to establishing net-based or local learning communities among Computer Science students. Students can reason about their programs in teams, and they get a concrete common space within the realm of a program. Also, exciting competitions can be arranged, like which team is the first one to uncover the secret of a program.

The bottom-up approach of path testing is not only its restriction. Paths discovered and experienced by different students, can serve as crucial steps for a holistic view of the algorithm under study.

Finally, while the system can support the learner by pointing out multiple paths of execution and adjusting the level of detail, it should probably be extended by a particular instructing tool. This tool could open up the interaction by motivating the students to make use of system's capabilities and the whole path testing approach.

References

1. Aho, A., Sethi, R., Ullman J.: Compilers, Principles, Techniques, and Tools. Addison Wesley, 1986.
2. Beizer, B.: Software Testing Techniques, 2nd edition. New York, Van Nostrand Reinhold, 1990.

3. Ben-Ari, M., Myller, N., Sutinen, E., Tarhio, J.: Perspectives of Program Animation with Jeliot. In this volume.
4. Brown, M.: Algorithm Animation. ACM distinguished dissertation, MIT Press, 1988.
5. Cornett S.: Code Coverage Analysis. *http://www.bullseye.com*, seen 08/01.
6. Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., Vanninen, P.: Animation of user algorithms on the Web. In Proc. 1997 Symposium on Visual Languages, 1997, 360–367.
7. Horspool, N.: Practical fast searching in strings. Software Practice & Experience 10, 1980, 501–506.
8. Hundhausen, C.: Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course. Doctoral Dissertation (Tech Rep. No. CIS-TR-99-07), Dept. of Computer and Information Science, University of Oregon, 1999.
9. Järvinen, E. M.: The Lego/Logo Learning Environment in Technology Education: An Experiment in a Finnish Context. Journal of Technology Education 9(2), 1988, 47–59.
10. Korhonen, A.: Algorithm Animation and Simulation. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2000.
11. Korhonen, A., Malmi, L.: Algorithm Simulation with Automatic Assessment. In Proc. ITiCSE '00, 5th Annual SIGCSE/SIGCUE Conference on Innovation and technology in computer science education, ACM, 2000, 160–163.
12. Korhonen, A., Malmi, L., Saikkonen, R.: Design Pattern for Algorithm Animation and Simulation. In E. Sutinen (Ed.): Proceedings of the First Program Visualization Workshop, University of Joensuu, Department of Computer Science, 2001, 89–100.
13. Lavonen, J., Meisalo, V., Lattu, M.: Visual programming: basic structures made easy. In E. Sutinen (Ed.): Proceedings of the First Program Visualization Workshop, University of Joensuu, Department of Computer Science, 2001, 163–177.
14. Matrix homepage: *http://www.cs.hut.fi/Research/Matrix/*
15. Pólya, G. How to Solve It? Princeton University Press, 1973.
16. Price, B. A., Baecker, R. M., Small, I. S.: A Principled Taxonomy of Software Visualization Journal of Visual Languages and Computing 4(3), 1993, 211–266.
17. Stasko, J., Badre, A., Lewis, C.: Do algorithm animations assist learning: An empirical study and analysis. In Proc. INTERCHI '93, Conference on Human Factors in Computing Systems, Amsterdam, The Netherlands (1993) 61–66.
18. Stern, L., Sondergaard, H., Naish, L.: A strategy for managing content complexity in algorithm animation. In Proc. ITiCSE '99, 4th Annual SIGCSE/SIGCUE Conference on Innovation and technology in computer science education, ACM, 1999, 127–130.
19. Vygotsky, L. S.: Mind in Society: The Development of Higher Psychological Processes. (M. Cole, V.J. Steiner, S. Scribner, E. Souberman, eds.) Harvard University Press, 1978.

Hypertextbooks: Animated, Active Learning, Comprehensive Teaching and Learning Resources for the Web[1]

Rockford J. Ross[1] and Michael T. Grinder[2]

[1] Computer Science Department,
Montana State University
Bozeman, MT 59717, USA
ross@cs.montana.edu

[2] Computer Science Department,
Montana Tech of the University of Montana
Butte, MT 59701, USA
grinder@cs.montana.edu

Abstract.

Computer-generated visualizations have been used in computer science education for many years, most notably in the form of algorithm animations. Although appealing and often useful, the anecdotal evidence is that these visualizations are seldom used in the classroom. There are many reasons for this, including platform dependence, cumbersome installation and maintenance procedures, and—perhaps most influential—a lack of integration with other course materials. Hypertextbooks provide one solution to these problems. Designed as complete teaching and learning resources for the web, hypertextbooks incorporate many features for teaching and learning that vastly extend the capabilities of traditional textbooks. Along with traditional textual presentations of the material to be learned, hypertextbooks allow for different learning paths through the material for different learning needs, an abundance of pictures and illustrations, video clips where helpful, audio, and—most importantly—interactive, active learning visualizations of key concepts. In this paper we discuss the hypertextbook concept by way of the hypertextbook project currently underway at Montana State University.

1 Introduction

Visualizations play a key role in providing insights into important concepts. Computer-based visualizations have been applied to many areas of science and engineering, enhancing our understanding of molecular structures, mysteries of the universe, predator-prey relationships, and many other science, engineering,

[1] Support for some of the work described here has come from the National Science Foundation, grant numbers NSF-0088728 and NSF-0088934.

and sociological phenomena. Most visualization software has been oriented towards advancing research. Less attention has been paid to the development of visualization software for education.

At first glance, it might appear that visualization systems developed for research would be equally applicable to education, but that is not the case. Although demonstrations of advanced visualizations can be helpful in the classroom, there is a vast difference between a visualization intended for an expert, who already understands the field well and knows what patterns to look for in a visualization (and what those patterns might mean), and one intended for a novice who does not understand the field well and is using visualization software to help learn the field.

Computer science educators have developed a number of visualizations of computer science concepts for education since the early days of the discipline [1, 14,18,16,9]. Most of these, not unreasonably, are algorithm animations (see, for example, [5,12,15])—visualizations that show the steps of an algorithm, such as quicksort, in action. Educators have always struggled to convey the dynamics of an algorithm in a lecture; the artistic and acting capabilities required for an illuminating presentation at a whiteboard elude most instructors. It is also difficult to avoid mistakes, and it is a struggle to back up in such a live presentation to answer student questions about what happened a few steps earlier. Finally, when students walk out of the classroom they leave the dynamic presentation behind. Notes they might have taken, being inherently static, are of little use in recapturing the dynamic information of the lecture.

It is no wonder, then, that the idea of dynamic, computer-based visualizations of key computer science concepts is so appealing to educators. Done properly, they are error free, repeatable, easy to reverse in answer to questions, usable for study outside of the classroom, and readily available to both instructors and students. In spite of their appeal, however, it is well known that visualization software for computer science education is not widely used in instructional settings. In the rest of this paper we discuss the reasons for this paradox and present one remedy: the hypertextbook.

2 Why Educational Visualization Software Is Underused

What is it that keeps visualization software systems—even good ones—designed to aid teaching and learning at bay? In [3] we examine this question in depth. Essentially, there are four parts to the answer to this question: (1) platform dependence, (2) installation and maintenance chores, (3) demands on faculty time, and (4) a lack of courseware integration.

The problems associated with platform dependence and installation and maintenance chores are self-evident. These problems can be overcome with well-designed visualization systems designed to run as applets on the web. Such applets are platform independent by definition, and they require no installation or maintenance on the part of the users. However, web-based visualization applets still require time on the part of instructors to locate, evaluate, learn, and teach in preparation for effective classroom use, a process that must be repeated for each desired visualization. Since most such applets deal with single concepts and are therefore useful for only one or two lectures, this is time that most overburdened faculty members are unable to invest. Finally—and most importantly—there is the problem of courseware integration. An individual applet acquired from the web for visualization purposes is unlikely to blend well with traditional course resources or other applets. Terminology may differ, the implementation of the concept may not closely match that presented in the course textbook, and it may be difficult to decide where to schedule the presentation and use of the applet in the course. Furthermore, individual applets retrieved from the web seldom provide comprehensive active learning experiences for students.

On one hand, then, it is no wonder that visualization software designed for education is underutilized. On the other hand, a solution to this problem is evident. Teaching and learning resources need to be developed that are platform independent and that incorporate active learning visualization applets as a seamless part of the whole. This brings us to the concept of the hypertextbook.

3 Hypertextbooks

A *hypertextbook* is a comprehensive, web-based teaching and learning resource that is intended to augment or supplant a traditional textbook for an academic subject. For example, a hypertextbook on the theory of computing would be a complete, web-based resource for teaching and learning the theory of computing. Hypertextbooks extend the capabilities of traditional textbooks tremendously in that, beyond mere textual presentations and static illustrations, they can also incorporate video clips, audio files, and active links to other material on the web. They can also be arranged (through the use of hyperlinks) to accommodate various teaching/learning needs and styles. Most unique, though, is their capacity for including active learning modules in the form of interactive applets that visualize important concepts and engage students in exploratory learning. In most cases, the visualizations are animated. That is, the concept or model being visualized changes over time in response to various stimuli; we often refer to such visualizations as *animations*. It is this capacity that web-based hypertextbooks have for the incorporation of active learning animation applets that we consider at length in this paper.

The ensuing discussion of hypertextbooks is based on our own work-in-progress on a hypertextbook we call *Snapshots of the Theory of Computing*, or just *Snapshots* for short. The title *Snapshots* reflects the fact that the hypertextbook is being made available in parts (snapshots), as each part becomes ready.

We have reported on *Snapshots* before [2,3]. Here we present the hypertextbook concept from a different point of view. As the design of *Snapshots* is discussed, we will highlight points that we have found to be important in the construction of a hypertextbook, both from pedagogical and practical points of view. For ease of reference, we will number these points.

It is important to note at the outset that none of the "important points" we list have actually been verified by us through formal studies with hypertextbooks in a teaching and learning environment; we have not yet had opportunity to conduct such studies[2]. Instead, the points listed come from our own experience, discussions with colleagues, presentations by cognitive psychologists, and the literature (for example, [15]).

It is equally important to acknowledge that many of the points we highlight—even though we arrived at most of them independently—are not unique to our work. The use of hypertext and hypermedia in teaching and learning has been investigated over the course of a number years (as an example, see [11]). The idea of constructing hypertext teaching and learning resources with integrated visualizations and animations is also, of course, not unique to our project. In [6], for example, eleven "design principles for effective web-based software visualizations which cover teaching requirements, sustainability, ease of use, and remoteness" are discussed based on the work of the authors, who build on earlier work [7]. On the other hand, we know of no other work currently in progress that captures the essence of our hypertextbook project, which focuses on the inclusion of comprehensive, integrated, and animated active learning applets.

With these acknowledgements to the substantial work of others, we (for the sake of brevity) will make few more references to the literature in presenting the list of issues we have found to be important in the design of a hypertextbook.

3.1 The Hypertextbook Cover

Figure 1 provides a view of the "cover" of *Snapshots* as it appears when viewed in the Netscape web browser[3]. This is the home page of the web that makes up the *Snapshots* hypertextbook and thus serves as the portal to the book. One will notice that this page does indeed appear similar to the cover of a traditional textbook. We have attempted to make it attractive, uncluttered, and functional, which leads us to our first points.

Point 1. The cover, or portal, of a hypertextbook should be attractive and/or intriguing, thus inviting readers to explore further.

Point 2. A hypertextbook, from the cover on, should have a familiar and professional "textbook" look to it, so that students feel comfortable using it as their main class learning resource. (As time goes on and hypertext materials become more prevalent, it is likely that such visual relationships to traditional textbooks will become unnecessary.)

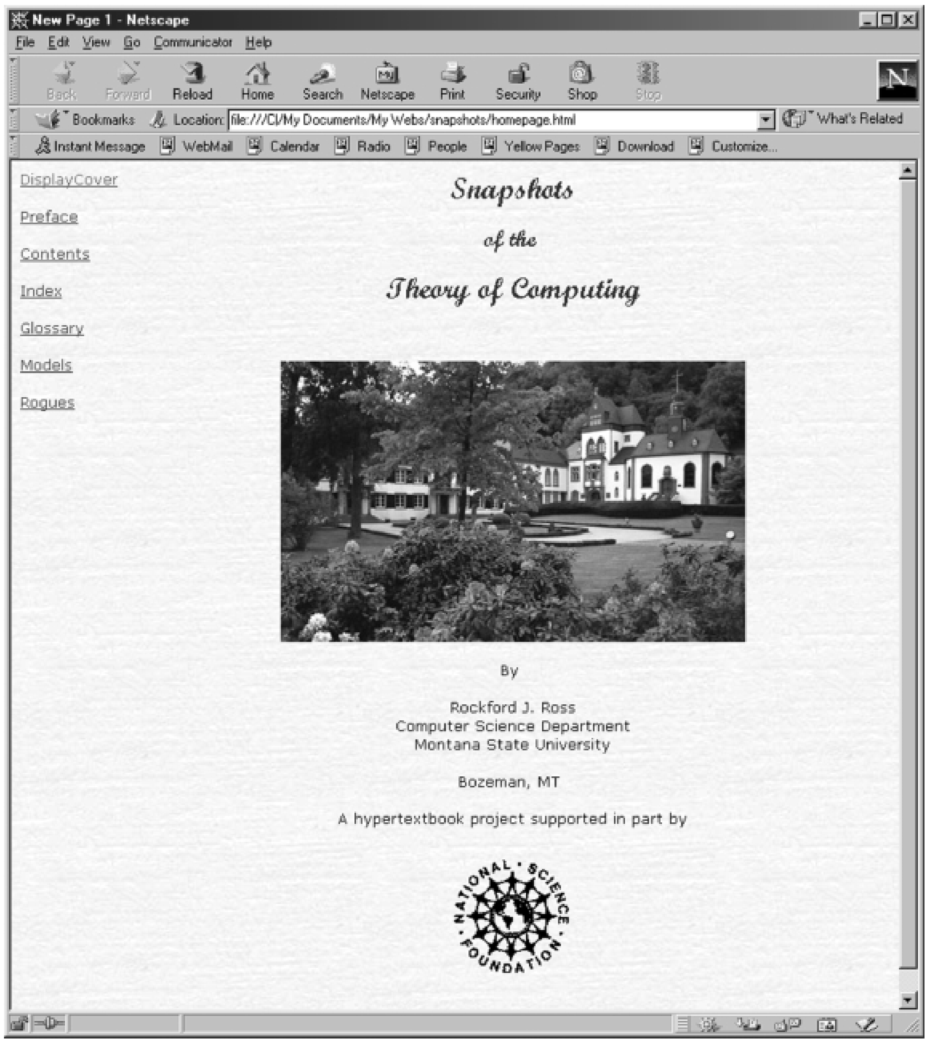
Point 3. A hypertextbook should be uncluttered, yet functional. There is a strong temptation to make liberal use of the "bells and whistles"

[2] This is the classical "chicken and egg" problem; such studies would help in the design of a hypertextbook, but one needs a hypertextbook to conduct the studies.

[3] It is, unfortunately, not possible to reproduce colors here. Where important we will explain the color schemes in the figures.

available for web page development in a hypertextbook (flashing symbols, odd fonts, animated images, and so forth), most of which only confuse the learner and detract from the learning experience.

We tried a number of different designs before recognizing the importance of points 1 through 3.


New Page 1 - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: file:///C|/My Documents/My Webs/snapshots/homepage.html
What's Related
Instant Message WebMail Calendar Radio People Yellow Pages Download Customize...
DisplayCover
Preface
Contents
Index
Glossary
Models
Rogues

Snapshots

of the

Theory of Computing

By

Rockford J. Ross
Computer Science Department
Montana State University

Bozeman, MT

A hypertextbook project supported in part by

NATIONAL SCIENCE FOUNDATION

Fig. 1. The cover of the *Snapshots* hypertextbook

As seen in figure 1, the main entry points to the hypertextbook are listed as links along the left side of the cover. This left margin with its links appears on every page of the hypertextbook, giving students easy access to the important parts of the book: cover page, preface, contents, index, glossary, standalone versions of the active learning visualization applets (the "Models" link) used in the book, and a list of the book's contributors (the "Rogues" link). Fonts, page backgrounds, and color schemes are also used consistently throughout the book.

> **Point 4.** A hypertextbook should maintain a consistent structure throughout. Standardized page design, common font usage, and consistent application of color schemes are all important.

3.2 The Hypertextbook Structure

The web provides opportunities for structuring a hypertextbook in ways that traditional textbook authors can only dream about. As examples, a hypertextbook can be organized—through the use of hyperlinks—to have different learning paths (through the same material) for different learning styles or for different levels of educational maturity. In *Snapshots* we have chosen the latter approach, creating paths through the book that cater to novices, intermediate learners, and advanced students, respectively. From the "Preface" link in the left margin one can get to a description of the organization of *Snapshots* which displays the illustration given in figure 2.

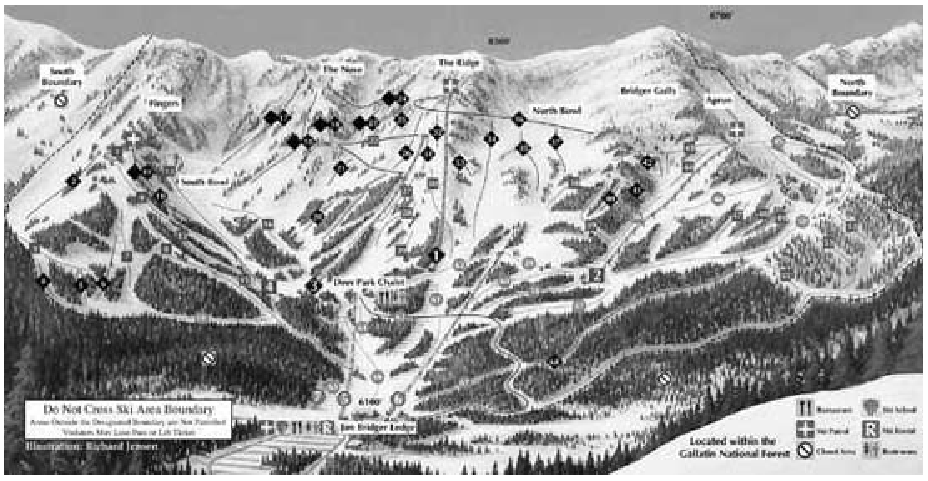
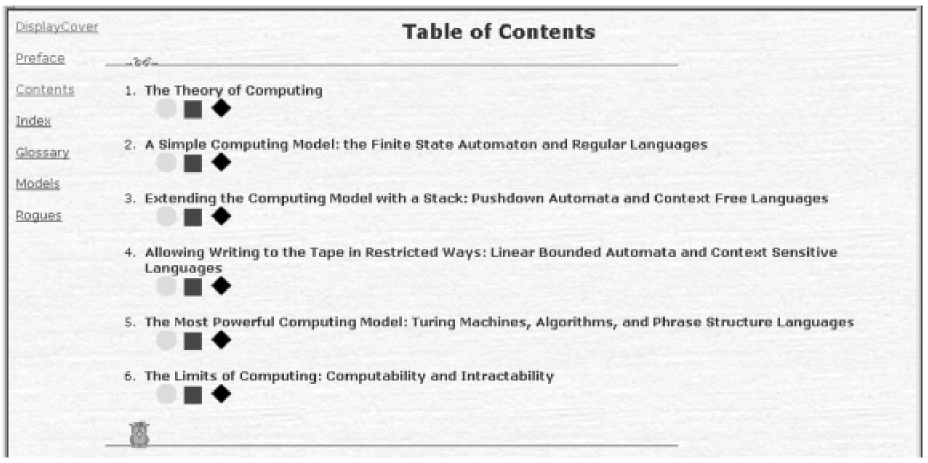


Fig. 2. The ski slope organizational model of *Snapshots*

Figure 2 is a copy of a ski trail map for Bridger Bowl, a ski destination in the Rocky Mountains near Bozeman, Montana, USA. One can see that there are many trails that start at various high points on the mountain. All trails are marked with international symbols to make it possible for skiers to choose the appropriate way down based on their abilities. Green circles mark the easiest routes, blue squares the intermediate paths, and black diamonds the challenging trails. All lead to the same goal: the lodge at the bottom.

The ski slope model is our inspiration for the organization of *Snapshots*. We are designing the book, as noted, to address the needs of beginners, intermediate learners, and advanced students. We mark the different ways through the book with the same international symbols used on ski slopes: a green circle for beginners, a blue square for intermediate learners, and a black diamond for the more advanced. All paths lead to the same goal: an understanding and appreciation of the theory of computing. This structure makes the book usable across the curriculum.

Point 5. A hypertextbook should make use of hyperlinks to organize the material in ways appropriate for different learning needs and/or different learning styles to make it as flexible and useful as possible.



DisplayCover
Preface
Contents
Index
Glossary
Models
Rogues

Table of Contents

1. The Theory of Computing
   ● ■ ◆
2. A Simple Computing Model: the Finite State Automaton and Regular Languages
   ● ■ ◆
3. Extending the Computing Model with a Stack: Pushdown Automata and Context Free Languages
   ● ■ ◆
4. Allowing Writing to the Tape in Restricted Ways: Linear Bounded Automata and Context Sensitive Languages
   ● ■ ◆
5. The Most Powerful Computing Model: Turing Machines, Algorithms, and Phrase Structure Languages
   ● ■ ◆
6. The Limits of Computing: Computability and Intractability
   ● ■ ◆

Fig. 3. The contents page of *Snapshots*

To see how this works, look at figure 3, which is a clip of the page reached by selecting the "Contents" link in the left margin. Notice that beneath each chapter title the three international symbols appear: the (green) circle, the (blue) square, and the (black) diamond. These symbols have associated hyperlinks that lead the user to a more detailed table of contents for that chapter based on the level selected. From the more detailed table of contents a student can then begin to learn about the topic of the chapter by clicking on the desired section of the chapter (each of which is also a hyperlink). The pages for each topic are marked at the top by the appropriate international symbol to let the students know whether they are on the appropriate track.

Following the green circle route leads a student through a very intuitive presentation of the topic being studied, with lots of examples and liberal use of the animated, active-learning applets that we have designed for the theory of computing. The blue square track also provides an intuitive introduction to the topic, but with fewer examples involving the active-learning applets and a greater reliance on mathematical notation. The black diamond approach incorporates only a few examples that use the active learning applets and resorts to formal mathematics throughout.

3.3 Animated, Active Learning Applets for Hypertextbooks

At this point we can finally discuss the central feature of the *Snapshots* hypertextbook—animated, active learning applets of the key concepts of the theory of computing. It might seem that these could be discussed in isolation, but doing so would obscure one of the most important issues of hypertextbook design: applets designed for use in a hypertextbook have substantially different requirements than those designed for standalone use. They also take much more time to create. An old subjective metric [8] states that if a software system designed for personal use takes time N to complete, the same system designed for use by others will take time $3N$, and if it is also to be integrated with another system (e.g., a hypertextbook) it will take time $9N$. Our experience certainly lends credence to this observation.

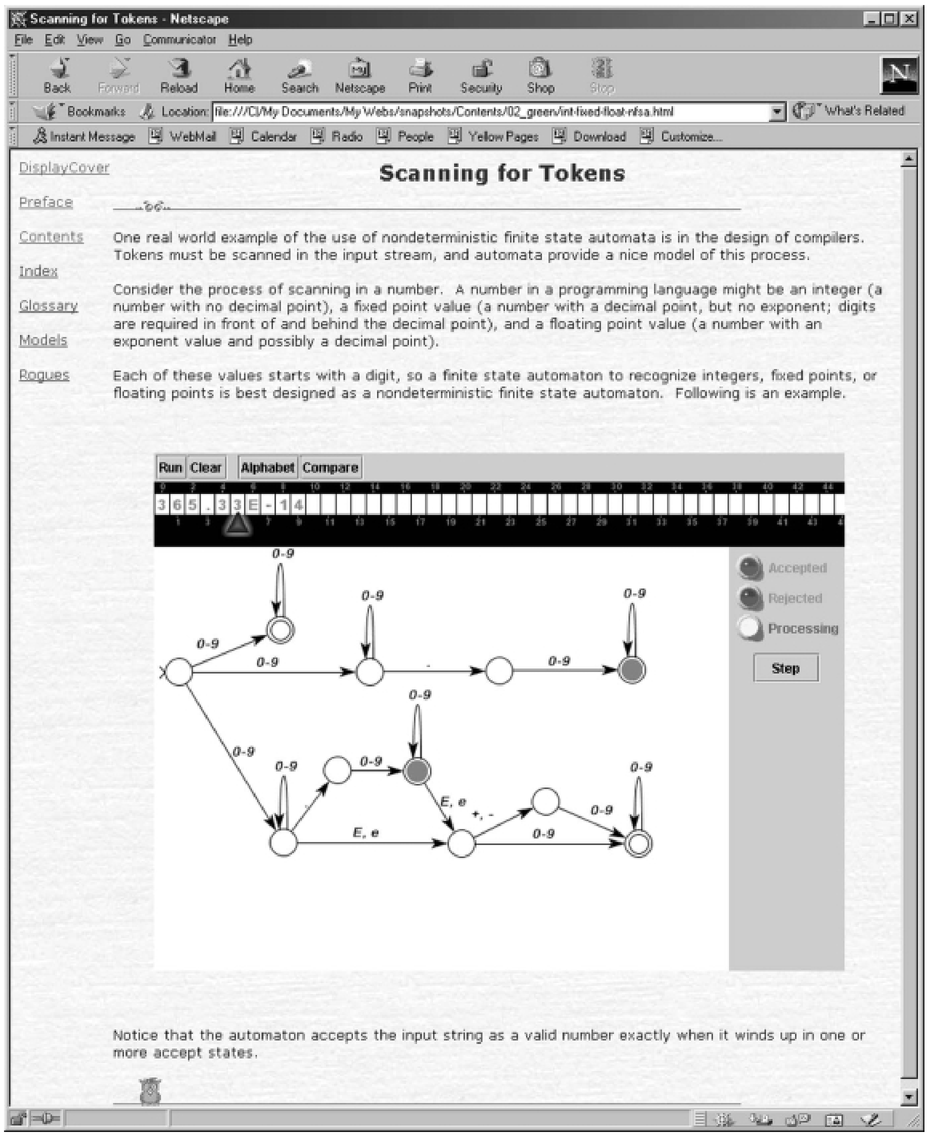
> **Point 6.** Creating animated, active learning applets that integrate well with each other and fit seamlessly into a hypertextbook takes appreciably more time than creating standalone applets for identical topics.

By way of introduction, look at figure 4. This is a page that appears towards the end of a section in *Snapshots* that discusses nondeterministic finite state automata on the green track. Notice that the applet is embedded directly in the text of this page. We have discovered that opening a new window for an applet is distracting to the student, in that it causes focus to be shifted away from the discussion in the text. It is also confusing in that students are not sure when a newly opened window should be closed, nor is it clear when or how to return to the text. On the other hand, standalone versions of the applet should also be available for students to use in their own explorations (this feature is provided in *Snapshots* through the "Models" link in the left margin).

> **Point 7.** Applets that are to be used in examples in a hypertextbook should appear in line and not be displayed in new windows.

> **Point 8.** Applets used in a hypertextbook should also be available (in an appendix, for instance) in standalone mode for arbitrary student or instructor use.

Figure 4 shows the automaton applet preloaded with an example nondeterministic automaton for determining whether an input string is a valid integer, fixed point, or floating point number. Configuring the applet for this particular use is, of course, the responsibility of the hypertextbook author.


Scanning for Tokens - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: file:///C|/My Documents/My Webs/snapshots/Contents/02_green/int-fixed-float-nfsa.html
What's Related
Instant Message WebMail Calendar Radio People Yellow Pages Download Customize...
DisplayCover
Preface
Contents
Index
Glossary
Models
Rogues
Run Clear Alphabet Compare
3 6 5 . 3 3 E - 1 4
Accepted
Rejected
Processing
Step
0-9
.
E, e
+, -

Scanning for Tokens

One real world example of the use of nondeterministic finite state automata is in the design of compilers. Tokens must be scanned in the input stream, and automata provide a nice model of this process.

Consider the process of scanning in a number. A number in a programming language might be an integer (a number with no decimal point), a fixed point value (a number with a decimal point, but no exponent; digits are required in front of and behind the decimal point), and a floating point value (a number with an exponent value and possibly a decimal point).

Each of these values starts with a digit, so a finite state automaton to recognize integers, fixed points, or floating points is best designed as a nondeterministic finite state automaton. Following is an example.

Notice that the automaton accepts the input string as a valid number exactly when it winds up in one or more accept states.

Fig. 4. An embedded finite state automaton applet in *Snapshots*

Point 9. A special software tool must be provided with each applet that allows an author to configure that applet properly for each appearance of that applet in the hypertextbook. For instance, if the applet in question is to illustrate a particular finite state automaton in an example at some desired point in the hypertextbook, the author must be able to configure the applet to start up with the desired automaton at that point.

Active learning applets should have a number of special features that aid learning. One is a feature that can lead a student through an animation a step at a time with little or no required intervention and with accompanying explanations. This feature is important when examples of a new topic are encountered for the first time. In the case of the finite state automaton applet, for instance, this would mean that the applet would be preloaded with both an automaton and an input string. Each time the student presses the "Step" button, the automaton would be shown consuming the current input symbol, changing states, and moving the input head to the next input symbol. An explanation of this step would appear simultaneously in another pane of the applet window.

A second feature of applets intended for active learning is one that gives students more responsibility for directing the animation. Again using the finite state automaton applet as an example, in this instance an automaton might be preloaded into the applet by the author, but the student would then provide his or her own input strings and run the automaton to see whether the strings were accepted or rejected.

A final necessary feature for active learning applets is one that assigns complete responsibility to the student (for example, when an applet is used in an exercise). In the case of the finite state automaton applet, the student should be able to create and modify an automaton arbitrarily within the applet in solving a given exercise or for independent exploration.

Other important features include the capability to back up in an animation (so that the student can explore puzzling aspects of the animation), and an option to set the animation either to proceed one step at a time under student control or to set it to run automatically with an accompanying method (e.g., a slider bar) for controlling the speed of automatic execution of the animation.

Point 10. Active learning animation applets intended for use in a hypertextbook must provide a wide range of control to the student, from virtually no control (so that a new concept can be explained a step at a time by the author), to intermediate control (so that a predetermined example can be explored by a student controlling the animation of that example), to complete control (so that a student can construct and control an animation of a concept from scratch in an exercise). It has been shown that the more involved a student is in creating an animation, the better the student learns [17,15].

Point 11. Active learning applets should provide capabilities for a student to control each step in an animation or to set the animation to run continuously; in the latter case, there should be a mechanism that allows the student to adjust the speed of automatic animation.

Point 12. Active learning applets should allow a student to back up arbitrarily far in an animation to review or retry certain steps.

In figure 4 the nondeterministic automaton is shown part way through the processing of the input string `365.33E-14`. There are three nondeterministic branches in this automaton that check whether the input string is an integer, fixed point, or floating point number, respectively. The current states are shaded (with a red disk). As an input symbol is consumed, a state transition is shown in that the red disks move smoothly from their current states simultaneously across the appropriate transition edges to the next states (if there is no corresponding transition from a state, the red disk turns gray and then disappears). The input head also moves to the next input symbol. State transitions can be controlled a step at a time through the "Step" button or set to run automatically. Rudimentary sound effects accompany these actions; there are different sounds for state transitions, string acceptance, and string rejection that draw attention to these activities and distinguish them from one another. One can see that the graphical version of the automaton reflects closely the form of automaton models found in traditional textbooks.

Point 13. Animated, active learning applets should provide smooth transitions between images, or states, in the animation. Students can then see more easily how the step being animated occurs.

Point 14. Sound should be used where appropriate in active learning applets. Sound can give important clues to an animation, which today's students are accustomed to utilizing, for example, while playing computer games, and even while interacting with general software systems (e.g., operating systems).

Point 15. Models animated in active learning applets should not deviate in appearance substantially from their traditional visual representations unless there are sound pedagogical reasons for a change.

3.4 Integrating Applets in Hypertextbooks

The repertoire of applets designed so far for *Snapshots* includes those needed for a complete first chapter of a traditional theory book on finite state automata, regular grammars, and regular expressions. In addition to the finite state automaton applet shown in figure 4, there is a context free grammar applet that allows an author to illustrate arbitrary context free grammars (and hence arbitrary regular grammars) and animate derivations with these grammars. Students can use this applet in exercises to create grammars and to construct derivations of arbitrary strings in those grammars.

There is also a regular expression grammar that an author can use to demonstrate the construction of a regular expression for a provided regular set. It also allows a student to construct regular expressions.

Finally, there is a program animator that animates (Pascal) programs. It can be used to demonstrate implementations of various algorithms related to the theory, such as how a finite state automaton can be implemented as a program, how a regular expression is converted to a finite state automaton, and so forth. (in this instance, Pascal serves as a nice pseudo-language). These applets are not illustrated here for lack of space, but can be found at [13].

The finite state automaton, context free grammar, and the regular expression applets have been designed to work together to provide feedback to a student completing exercises using these applets[4]. Using known properties of finite state automata, the automaton applet incorporates an algorithm that checks a student-constructed automaton for accuracy. To accomplish this, the hypertextbook author providing the exercise also gives a correct finite state automaton (hidden from the student) for the exercise. The language of the correct automaton is then compared with the language of the student's automaton. If the two are equal, the student is congratulated. If the two languages are not equal, the applet reports this to the student along with a sample string that the student's automaton either accepts or rejects in error.

Using other known algorithms that convert regular expressions and regular grammars to equivalent finite state automata the same technique is applied in the regular expression and grammar animation applets to provide similar feedback to a student. Again, the author must provide a correct regular expression or grammar, respectively, when creating exercises with these applets. Then the conversions to equivalent finite state automata of the correct version and the student version are made, the language comparisons completed as above, and feedback provided to the student.

Point 16. Active learning models included in a hypertextbook must be designed to interact with each other as appropriate.

3.5 Decoupling a Model from Its Description

This brings us to our final point, and perhaps the most important one we make.

Point 17. The data structure that describes a model (e.g., a finite state automaton) must be independent of its graphical representation in an applet.

This last point is easily overlooked. It is the reason why many standalone applets cannot readily be extended, or incorporated into other resources, such as hypertextbooks. Without an underlying representation for each model that is independent of its graphical presentation, integrating applets to work together (as described in the previous section) would be prohibitively difficult. With *Snapshots* we have implemented the most logical solution to this problem by designing for each model (finite state automaton, grammar, and regular expression) an eXtensible Markup Language (XML) definition. Not only does this allow the various models to be integrated with each other, but it also provides for the development of different graphical representations of the models as desired, and it allows the models to be treated in a fashion that is now standard on the web.

Briefly, this works as follows. Consider the definition of a finite state automaton. This definition can be formally specified in XML so that all of the components of a finite state automaton—the states, the input alphabet, the

[4] Much of this work is in progress at the time of this writing.

transition function, and the accept states—are well defined. Each different finite state automaton is formulated with the same XML structure, but with different values in the data fields (e.g., the fields that represent the number of states, the actual input alphabet, and so forth). In our case, where pedagogy is key, there may also be other parts to the definition not normally associated with the theoretical definition of a finite state automaton, such as a field that accompanies each state that describes what that state "remembers," or a particular input string on which the automaton should be started when it is initialized. Along with the XML file for a finite state automaton, a Document Type Definition (DTD) file is provided that specifies the rules for constructing a proper finite state automaton in XML. Thus, programs that process the XML file can check it against its DTD to see that the file represents a correctly structured finite state automaton before actually processing the XML file.

The key point is that the XML file for a finite state automaton contains only its definition, not any information about how the automaton is to be displayed. Thus, it is up to the program processing an automaton XML file to decide how to display the automaton. The illustration in figure 4 represents one way to display a finite state automaton from its XML file. The same automaton could be displayed as a table from the same XML file, since no information is included in the file about how to display or animate the automaton. On the other hand, since all of the information about the automaton is in the XML file, it is possible to display the automaton consistently in one of these various forms, and even to animate it as desired.

Consider also what is done when an author creates an exercise for the hypertextbook using an exercise creation tool (see point 9). Suppose this exercise requires a student to construct a finite state automaton to recognize a particular regular language. In developing the exercise, the author provides both the written specifications of the automaton for the student and a correct automaton. This correct automaton is automatically converted to its XML form by the exercise construction tool and stored as part of the exercise (hidden from the student). The student completing the exercise then attempts to construct a correct automaton within the finite state automaton applet. Behind the scenes, the applet converts the student's automaton to its respective XML form. When the student then clicks on a "submit" button, the applet invokes an algorithm that compares the correct automaton against the student's automaton using a well-known algorithm for determining whether two finite state automata recognize the same language. This algorithm is easy to implement because of the consistent representation of both automata as correct and consistent XML files (each based on the same DTD).

The context free grammar and regular expression models in *Snapshots* also have appropriate, consistent XML representations. This makes implementation of standard conversion and checking algorithms (e.g., algorithms for checking whether a student-constructed regular expression is correct or not, or for converting a regular expression to a finite state automaton) in the active learning applets that animate these concepts straightforward as well.

Based on the XML definitions for the different models, work is continuing as of this writing on extending the active learning applet library of *Snapshots* to include animations of the standard conversion algorithms from nondeterministic finite state automata to equivalent deterministic versions, from deterministic finite state automata to their minimal form, from regular expressions to finite state automata (and vice versa), and from regular grammars to finite state automata (and vice versa). Finally, animations of the applications of the theory—such as the pumping lemma for regular languages, the Myhill-Nerode theorem, and others—are in the plans. Together, these animations will support a comprehensive first chapter in *Snapshots* on finite state automata, regular grammars, and regular expressions, replete with animated, active learning applets designed to help students of many different abilities and backgrounds come to an appreciation and understanding of the theory of computing.

4 Summary

In this paper we have discussed some of the problems that have precluded the widespread use of standalone educational visualization software systems in the computer science curriculum. We have then proposed one solution to these problems: hypertextbooks that are designed for the web and are thus platform independent, turnkey systems that can be used as the primary teaching and learning resource in a course. Such hypertextbooks can (and should) incorporate animated, active learning applets in a seamless fashion, so that students use them as a matter of course. Finally, we have listed a number of important points that we have learned in the course of our efforts to construct hypertextbooks in the Webworks Laboratory [13] at Montana State University.

There are a number of researchers working on visualization and animation software for education, including animations of theory concepts (see, for example, [10]). However, we know of no other hypertextbook projects of the scope described here (one project worth a look is [4]). We hope that our work will encourage others to begin similar projects. Patience is required. The writing of a textbook is a big job in any case. Designing a hypertextbook that addresses the differing needs of students and that incorporates active learning applets makes the job much more challenging. In the end we believe that students will profit.

References

1. 16mm color sound film. *Sorting Out Sorting*. 30 minutes, 1981.
2. Boroni, C. M., Goosey, F. W., Grinder, M. T., and Ross, R. J. A Paradigm Shift! The Internet, the Web, Browsers, Java, and the Future of Computer Science Education. In *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* (Mar. 1998), vol. 30, number 1, pp. 145–149.
3. Boroni, C. M., Goosey, F. W., Grinder, M. T., and Ross, R. J. Engaging Students with Active Learning Resources: Hypertextbooks for the Web. In *Thirty Second SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* (Mar. 2001), vol. 33, number 1, pp. 65–69.

4. Braune, B., Diehl, S., Kerren, A., and Wilhelm, R. Ganimal project: Ganifa—electronic textbook on generating finite automata. http://www.cs.uni-sb.de/GANIMAL/, 2001.
5. Brown, M. H. Zeus: A System for Algorithm Animation. In *Proceedings of the 1991 Workshop on Visual Languages* (Oct. 1991).
6. Domingue, J., and Muholland, P. An Effective Web Based Software Visualization Learning Environment. In *Journal of Visual Languages and Computing* (Oct. 1998), vol. 9, number 5, pp. 485–508.
7. Eisenstadt, M., and Brayshaw, M. *Understanding the Novice Programmer*. Erlbaum, Hilldsdale, NJ, E. Soloway and J. Spohrer, Eds., 1987, ch. An integrated textbook, video and software environment for novice and expert Prolog programmers.
8. Frederick P. Brooks, J. *The Mythical Man-Month*. Addison Wesley, 1975.
9. Greening, T., Ed. *Computer Science Education in the 21st Century*. Springer Verlag, 2000, ch. Shifting Paradigms: Teaching and Learning in an Animated, Web-Connected World, pp. 173–193. Invited chapter by Rockford J. Ross.
10. Hung, T., and Rodger, S. H. Increasing Visualization and Interaction in the Automata Theory Course. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* (Mar. 2000), vol. 32, number 1, pp. 6–10.
11. Jacobson, M. J., Maouri, C., Mishra, P., and Kolar, C. Learning with hypertext learning environments: Theory, design, and practice. In *Journal of Educational Multimedia and Hypermedia* (1996), vol. 5, number 3/4, pp. 239–281.
12. Naps, T. L., Eagan, J. R., and Norton, L. L. JHAVÉ — An Environment to Actively Engage Students in Web-based Algorithm Visualizations. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* (Mar. 2000), vol. 32, number 1, pp. 109–113.
13. Ross, R. J. *Webworks Laboratory Web Site*. http://www.cs.montana.edu/webworks.
14. Ross, R. J. Teaching Programming to the Deaf. *ACM SIGCAPH Bulletin 30* (Autumn 1982).
15. Stasko, J. Evaluating Animations as Student Aids in Learning Computer Algorithms. *Computers & Education 33*, 4 (1999), 253–278.
16. Stasko, J., Domingue, J., Brown, M. H., and Price, B. A., Eds. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.
17. Stasko, J. T. Using Student-Built Algorithm Animations as Learning Aids. In *Twenty-eighth SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* (Mar. 1997), vol. 29, number 1, pp. 25–29.
18. Thomas, D. A., Ed. *Scientific Visualization in Mathematics and Science Teaching*. Association for the Advancement of Computing in Education (AACE), 1995, ch. Visualizing Computer Science. Invited chapter by Rockford J. Ross.

Chapter 4
Graphs in Software Visualization

Introduction

Petra Mutzel[1] and Peter Eades[2]

[1] Vienna University of Technology,
Institute of Computer Graphics and Algorithms,
Favoritenstr. 9-11 E186,
A-1040 Wien, Austria,
`mutzel@ads.tuwien.ac.at`

[2] University of Sydney,
Basser Department of Computer Science, NSW
2006, Sydney, Australia,
`peter@cs.usyd.edu.au`

In a recent survey, Koschke [38,39] (see Chapter on Software Engineering) investigated the perspectives on software visualization of 83 researchers in the areas of software maintenance, reverse engineering, and re-engineering. One of the main results was that graphs have been identified as the most often used kind of visualization. Since in these areas graphs are generally computed by automatic analyses, tools are needed for laying them out automatically. Graph visualization and automatic layout are also important issues according to the survey by Bassil and Keller [2] on software visualization.

The discipline of designing algorithms and developing tools for automatic graph layout is called *graph drawing*. Software Visualization is an important application of graph drawing, yet not the only one. Other applications include the visualization of networks (e.g., social networks, computer networks), the visualization of processes (e.g., chemical reactions or business processes), or the visualization of data bases. Furthermore, graph drawing plays an increasingly important rôle in data mining, bioinformatics, or web-visualization.

Despite the wide variety in applications of graph drawing, some general aesthetic criteria have emerged. It is essential for the readability of a diagram that unrelated nodes do not intersect each other. Also intersections between nodes and unrelated edges should be avoided. According to the survey of Purchase [46], the avoidance of edge crossings is among the most important criteria. However, edge crossings cannot always be avoided. Graphs that can be drawn in the plane without edge crossings are called *planar graphs*.

Another important criterion for good readability is the avoidance of sharp edge bends. This problem vanishes when an orthogonal drawing style is used. Here, every edge is drawn as a sequence of horizontal and/or vertical line segments. The survey of Purchase [46] has shown that in orthogonal drawings the number of (90°-)bends should be small. A small number of bends tends to lead to a short total edge length and a small drawing area — these are other important aesthetic criteria. It is much easier to understand the structure of a graph in a drawing in which all edges are short. Moreover, if the drawing area is small, it is easier to identify objects on the screen due to the better resolution.

Additional aesthetic criteria are given by the specific applications. Often, the given data is hierarchical (such as, e.g., in a flow diagram), and this must be reflected in the drawing. Other criteria include display of symmetry or patterns.

Graph drawing algorithms can be classified into algorithms for general graphs and algorithms for hierarchical graphs. Typically, hierarchical graphs are directed and acyclic (called *DAGs*). Recently, a third category of algorithms for so-called *mixed graphs* developed. Mixed graphs consist of hierarchical and non-hierarchical parts.

In the following we will give a brief description of the most popular drawing algorithms for general and hierarchical graphs. Afterwards, we will investigate recent developments for topics that are mainly relevant to software visualization, such as scalability (handling large graphs including clustering, folding and viewing), interactive layout (incremental drawing, mental map, constraints), as well as recent progress in drawing of UML class diagrams.

The most popular drawing algorithms for general graphs are based on force-directed methods and on orthogonal methods for planarized graphs. Algorithms for hierarchical drawing concentrate on tree layout methods and the layered method suggested by Sugiyama et al. [52].

Force-directed methods. The idea of force-directed algorithms is to model a physical system and to compute its equilibrium. For this, the physical force model needs to be defined first. Then, coordinates for the nodes are computed that define a state of low total potential energy. Typically, the edges are drawn straight-line. For both steps many approaches have been suggested in the literature.

The forces typically depend on characteristics such as the node distribution, the edge lengths, the number of edge crossings, the node-edge distances, or the drawing area. A state of minimum energy can be approximated using iterative local search heuristics like simulated annealing, genetic or tabu search methods.

Force-directed methods are easy to understand and easy to implement. They work in two and three dimensions. For tree-like structures (very sparse graphs that are almost trees), such methods typically lead to nice drawings. Force directed methods, however, suffer from several drawbacks. The first is that the drawings often contain crossings that could be eliminated easily by visual inspection. Furthermore, most force directed methods produce straightline drawings, and as such do not deal well with large node sizes. Thirdly, most force directed methods are compute-intensive and not feasible for large graphs. Recently, various authors have suggested ideas to overcome these problems, e.g., [23,7,47, 47,54], at some cost to their simplicity. For non-tree like structures, the results of force-directed methods are — in many cases — not competitive with other methods. An overview of force-directed methods is given in [10] and [6]. The first method of this type has been introduced to graph drawing by Eades [17], who suggested a model based on springs.

Many software tools implement force-directed graph drawing methods. E.g., Graphlet, a toolkit for graph editors and graph algorithms [9] contains many different force-directed drawing methods.

Orthogonal methods based on planarization. The idea is to use the powerful algorithms for drawing planar graphs also for non-planar graphs. This is supported by the observation that graphs arising in practice are *almost* planar in the sense that only a few edges need to be deleted in order to obtain a planar graph. This observation directly leads to a simple planarization idea: In a first step delete the minimum number of edges such that the remaining graph is planar. Then draw the graph and reinsert the deleted edges while minimizing the edge crossings.

Some details are problematic: The first one is that deleting the minimum number of edges of a graph $G = (V, E)$ such that the remaining graph P is planar is an NP-hard optimization problem. However, it can be solved in practice by using efficient heuristics based on linear time planarity testing algorithms. The second problem is that the user may recognize the deleted and re-inserted edges in the final drawing. Therefore, the removed edges are inserted before the planar graph P is drawn. For this, a (planar) combinatorial embedding Π of P is computed. Formally, a *(planar) combinatorial embedding* Π of a planar graph P consists of a circular ordered list of the neighbours for each node in some planar drawing of P. A combinatorial embedding essentially fixes the faces of a planar drawing. The third problem is that re-inserting the edges into the planar graph P while minimizing the number of edge crossings is also NP-hard. It can be solved to optimality in linear time if only one edge needs to be inserted [27]. Therefore, a practical heuristic consists of an iterative method that inserts the edges one-by-one into the planar graph and substitutes the new crossings by new additional nodes.

After the reinsertion process, all the crossings have been substituted by artificial nodes, yielding the so-called *planarized* graph (which is planar, and re-substituting the artificial nodes by the crossings gives the original graph).

Now the planarized graph can be drawn by any planar drawing algorithm. The most popular drawing algorithms are based on network-flow computations. They originate from [53], in which Tamassia suggests a polynomial time algorithm for computing a bend-minimal orthogonal drawing of a graph G with bounded degree four when a fixed planar combinatorial embedding of G is given. The algorithm proceeds in two steps. First, the graph is transformed into a network in which a minimum cost flow is computed. From the flow, the angles between adjacent edges and the bends along the edges can be determined. This fixes a “shape” for each edge, and thus, of the entire drawing. In the second step, the task is to determine the lengths of the edge segments that compose the given shapes. The optimization goal here is to keep the total edge length, the maximal edge length, or the total drawing area small. Two-dimensional compaction is NP-hard, but there are many heuristics (e.g., [35]) and also an exact algorithm, which is able to solve most instances arising in practice very fast [37].

The bend minimization algorithm works only for graphs with node degrees bounded by four. Many efforts have been made to generalize the bend-minimization algorithm for general planar graphs [53,21,36]. It turned out that, indeed, drawings with fewer bends are smaller and look nicer. The number of bends in a drawing highly depends on the chosen combinatorial embedding. Even though bend minimization over the set of all combinatorial embeddings is NP-hard, two algorithms based on branch-and-bound and SPQR-trees are able to solve moderately sized instances of the problem to provable optimality [8,44].

There are very few software tools realizing the planarization method (e.g., AGD [28] and GDToolkit [25]) due to the difficult implementation effort. The AGD system is discussed in more detail in this chapter.

Layered methods for hierarchical graphs. Hierarchical structures appear, e.g., in data-flow diagrams, workflow diagrams, or activity charts. Formally, a directed graph G is called *hierarchical* if it does not contain a directed cycle. In drawings of hierarchical graphs, all the edges should point in one direction, e.g., from top to bottom or from left to right. By far most of the practical software used for laying out hierarchical data is based on the layering method suggested by Sugiyama, Tagawa and Toda in 1981 [52].

Their idea is to proceed in three steps. In the first step, the node set V of the directed graph $G = (V, A)$ is partitioned into k sets $V_1, \ldots, V_k$, called the layers of G, so that for each edge $(u, v) \in A$, $u \in V_i$, $v \in V_j$ with $i, j \in \{1, \ldots, k\}$ the condition $i < j$ holds. All nodes in the same set V_i, $i \in \{1, \ldots, k\}$, do finally get the same y-coordinate thus ensuring that all edges point into the same direction. In the second step the nodes within each layer are ordered so that the number of edge crossings gets small. In the final step, the y-coordinate of each layer and the x-coordinate of each node are computed.

For the first step various methods have been suggested. Topological sorting computes a layering with minimal height. The Coffman-Graham algorithm finds a layering of width at most w and height $h \leq (2 - 2/w)h_{\min}$, where $h_{\min}$ is the minimum height of a layering of width at most w [13]. Gansner et al. [22] have suggested an integer linear program (which can be solved in polynomial time) for finding a layering with the minimum total edge length in y-direction. Recently, Healy and Nikolov [30] have suggested an algorithm based on integer linear programming which attacks the problem of computing a layering subject to arbitrary constraints on the width and height.

The crucial step is the crossing minimization. Crossing minimization for a layered graph is NP-hard even for two layers one of which is fixed. Recently, an algorithm based on integer linear programming has been suggested which is able to solve instances of moderate size to provable optimality [29].

Before heuristics for crossing minimization can be applied, the graph needs to be transformed into a *proper hierarchy* in which each edge (u, v) with $u \in V_i$ and $v \in V_j$, $i + 1 < j$ is substituted by a path of edges $(u, w_1), (w_1, w_2), \ldots, (w_{j-i-1}, v)$, $w_l \in V_{i+l}$ for $l = 1, \ldots, j - i$. The heuristics are all based on the idea to successively consider two adjacent layers in which the ordering of the nodes in the first layer L_1 is already fixed whereas an ordering in the second layer L_2 is searched so that the number of crossings between the set of edges $\{(u, v) \mid u \in L_1, v \in L_2\}$ is minimized. The edges are drawn straight-line between two adjacent layers. Since also this *fixed 2-layer crossing minimization problem* is NP-hard, many heuristics for solving the problem have been suggested in the literature (see, e.g., [33,6]). Jünger and Mutzel [33] transformed the problem into the linear ordering problem which can be solved efficiently in practice using polyhedral combinatorics and branch-and-cut.

Alternative approaches [42,29] based on ideas for planarizing hierarchical graphs have been suggested, but so far they did not find their way into publically available software tools. An overview of hierarchical drawing methods is given, e.g., in [6]. The idea of the Sugiyama method can also be transferred to three-dimensional drawings (cone trees).

Sugiyama-type methods are available in almost all graph drawing software products; see, e.g., Graphviz [19] and aiCall [20]. The latter system is discussed in more detail in this chapter.

Tree drawing. A connected hierarchical graph containing no cycle (in the undirected sense) is called a *tree*. Trees usually represent hierarchies, and then the unique node with no predecessor is called the *root* of the tree. The most widely used algorithms for drawing a rooted tree such that the nodes of depth k in the tree are placed at a vertical distance of k below the root are based on a method by Reingold and Tilford [48] for drawing binary trees. In a binary tree, every subtree has at most two children subtrees, a left and a right one. The idea is to recursively draw the left and right subtrees independently in a bottom-up manner, then shift the two drawings in x-direction as close to each other as possible, and center the parent of the two subtrees one level up between their roots.

Drawings computed by this algorithm have the nice property that identical subtrees are drawn identically. Moreover, the width of the drawing is relatively small. In general, the drawings look aesthetically pleasing. The algorithm can be easily extended to general rooted trees. Most drawing tools contain tree drawing algorithms based on this idea.

Drawing of large graphs. The graphs arising from software visualization are often very large (having several thousands of nodes and edges). The classical graph drawing algorithms are, in general, not able to handle such huge data sets. The tree drawing algorithm is a notable exception since it can be implemented to run in linear time. Hierarchical algorithms are more problematic since already the layering of the graph can introduce a quadratic number of dummy nodes. However, in practice, this does not seem to be a bottleneck. The situation is different for planarization methods. Here, the bottleneck is the edge re-insertion phase, which needs time $O(r|V|)$, where r is the number of edges deleted in the first step. In the worst case, r can be quadratic in $|V|$. Force-directed methods suffer most with large graphs, but recent progress is promising, see, e.g., [24]).

In any case, it is difficult to make sense of many thousands of objects on a computer screen. Such problems can be resolved by fish-eye views or restriction to a small part of the graph in one window and a navigation tool for exploring the entire graph in another window. However, following single edges through the drawing can be very hard.

An alternative approach is to group node subsets in clusters, and only display the cluster node instead of all interior nodes. This can be done recursively. The idea is that the user starts with a small drawing containing cluster nodes that can be folded or unfolded interactively. One problem with this approach is the determination of the clusters. If they are not provided by the user, clusters are usually determined based on local properties (such as connectedness or the number of neighbours [3]). Another problem is the display of nodes and their cluster at the same time. Recently, a lot of progress has been made, e.g. in [51] for hierarchical graphs, in [47,40,5] for the orthogonal approach with planarized graphs, and in [31] for force-directed methods.

Tools focussing on drawing large graphs are, e.g., Tulip [1] and Pajek [4].

Interactive drawing. Koschke [38] also mentions that classical graph drawing methods use user-independent optimization criteria in order to generate the layout. In many cases, interactive approaches that allow users to help the system in producing nice drawings by giving hints to graph drawing algorithms are desirable.

Recent papers present ideas for user guided the optimization processes. This can be done by introducing visual constraints [49], by asking the user for his/her favourite drawings produced during the optimization process [32] or by manual changes during the process [15].

Another issue in Koschke's survey is "preserving the mental map". After adding or deleting nodes or edges, and calling the automatic layout algorithm, it can happen that the drawing changes significantly, even when this was not necessary at all. Some papers are investigating the question of preserving the essential properties of the layout, also called the *mental map*. Recent papers in this area include [45] for hierarchical graphs, [12] for interactive orthogonal graph drawing, and [31,14] for force-directed methods.

Tools specialized on interactive drawings include, e.g., Graphviz [45,19], Wilmascope [16], and GLIDE [49].

UML class diagrams. In UML class diagrams, class hierarchies in object oriented programs are displayed along with various association relations. While the class hierarchies are DAGs that should be drawn hierarchically, the associations should be represented as undirected edges. Such "mixed graphs" present a recent new challenge to graph drawing algorithms.

So far, the literature contains two different approaches. The first is to delete the non-hierarchical edges, use a hierarchical approach to draw the remaining graph, and then to re-insert the deleted edges [18]. One problem is that edge routing is a very hard problem. This approach works only well for graphs with relatively few association edges.

An alternative approach is based on the idea of orthogonal drawing of planarized graphs. However, during the planarization process one needs to make sure that the hierarchical structures are indeed displayed hierarchically in the final drawing. Also the planar orthogonal drawing algorithm need to be adapted accordingly [26].

Currently available software tools are described in [26] and [18]. The latter one is discussed in more detail in this chapter.

Conclusion. In his survey paper [39], Koschke states that the software visualization and the graph drawing communities are almost disjoint and that a collaboration of both communities would be desirable. The list of the most heavily used graph drawing tools in this survey contains obsolete tools that are not even maintained anymore, and does not contain new software that addresses some of the needs of the software visualization community.

As pointed out above, the needs of the software visualization community in terms of graph drawing tools are, at least partially, a subject of active investigation in the graph drawing community, and indeed, there are tools for problems such as drawing large graphs, interactive drawing, or drawing UML diagrams.

In the Dagstuhl meeting, ideas have been born to bring together the new XML-based graph exchange format GraphML [11] currently being developed in the graph drawing community and the GXL format [56,55] designed for software engineering. (GXL will be discussed in this chapter.) The ideas have been followed up by a discussion at the International Graph Drawing Symposium 2001 in Vienna. We hope that the two formats will merge into one. The Dagstuhl conference was a promising first step in bringing the two communities together, and hopefully, the articles in this chapter of the proceedings will enhance the collaboration.

For further reading, we suggest one of the books on graph drawing [6,34,50] and the yearly proceedings of the International Symposium on graph drawing, e.g., [41,43].

References

1. D. Auber. Tulip, a huge graphs visualization software. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 434–435. Springer-Verlag, 2002.
2. S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'2001)*, pages 7–17, Toronto, ON, 2001. IEEE.
3. V. Batagelj and A. Mrvar. Pajek – program for large network analysis. *Connection*, 21(2):47–57, 1998.

4. V. Batagelj and A. Mrvar. Pajek - analysis and visualization of large networks. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 474–475. Springer-Verlag, 2002.
5. G. Di Battista, W. Didimo, and A. Marcandalli. Planarization of clustered graphs. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 60–74. Springer-Verlag, 2002.
6. G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing*. Prentice Hall, 1999.
7. F. Bertault. A force-directed algorithm that preserves edge crossing properties. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 351–358. Springer-Verlag, 1999.
8. P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. In *WADS '97*, volume 1272 of *Lecture Notes in Computer Science*, pages 331–344, 1998.
9. F. Brandenburg and M. Himsolt. *Graphlet*. Available via "http://infosun.fmi.uni-passau.de/Graphlet/".
10. F.J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 1996.
11. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, page 497. Springer-Verlag, 2002.
12. S. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. Interactive giotto: An algorithm for interactive orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 303–308. Springer-Verlag, 1998.
13. E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.
14. J. Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Transactions on Computer-Human Interfaces*, 4(11):197–229, 1997.
15. H.A.D. do Nascimento and P. Eades. User hints for directed graph drawing. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, 2002.
16. T. Dwyer and P. Eckersley. Wilmascope interactive 3d graph visualisation system. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 440–441. Springer-Verlag, 2002.
17. P. Eades. A heuristic for graph drawing. *Congr. Numer.*, 42:149–160, 1984.
18. Holger Eichelberger and Jürgen Wolff v. Gudenberg. On the Visualization of Java Programs. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
19. J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz - open source graph drawing tools. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 480–481. Springer-Verlag, 2002.
20. Alexander A. Evstiougov-Babaev. Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*

21. U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 254–266. Springer-Verlag, 1996.
22. E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. In *IEEE Transactions on Software Engineering*, volume 19 (3), pages 214–229, 1993.
23. E. R. Gansner and S. C. North. Improved force-directed layouts. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '99)*, volume 1547 of *Lecture Notes in Computer Science*, pages 364–373. Springer-Verlag, 1998.
24. P. Gayer, M. Goodrich, and S. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In J. Marks, editor, *Graph Drawing (Proc. 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 211–221. Springer-Verlag, 2001.
25. GDToolkit. `http://www.dia.uniroma3.it/~gdt`.
26. C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. Automatic layout of uml class diagrams. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
27. C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '2001)*, pages 246–255, Washington, DC, 2001. ACM Press.
28. Carsten Gutwenger, Michael Jünger, Gunnar Klau, Sebastian Leipert, and Petra Mutzel. Graph Drawing Algorithm Engineering with AGD. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
29. P. Healy and A. Kuusik. The vertex-exchange graph: a new concept for multi-level crossing minimization. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1999.
30. P. Healy and N. S. Nikolov. How to layer a directed acyclic graph. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2002.
31. M. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '99)*, volume 1547 of *Lecture Notes in Computer Science*, pages 374–383. Springer-Verlag, 1998.
32. A. E. Jacobsen. Interaktion und Lernverfahren beim Zeichnen von Graphen mit Hilfe evolutionärer Algorithmen. Master's thesis, Institut AIFB, Universität zu Karlsruhe, 76128 Karlsruhe, Germany, 2001.
33. M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications (JGAA) (http://www.cs.brown.edu/publications/jgaa/)*, 1(1):1–25, 1997.
34. M. Kaufmann and D. Wagner, editors. *Drawing graphs: methods and models,* volume 2025 of *Lecture Notes in Computer Science Tutorial*. Springer, 2001.
35. G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Graph Drawing (Proc. 2000)*, LNCS. Springer Verlag, 2001.
36. G. W. Klau and P. Mutzel. Quasi–orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max–Planck–Institut für Informatik, Saarbrücken, 1998.
37. G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In G. P. Cornuejols, editor, *Integer Programming and Combinatorial Optimization (IPCO '99), Proceedings of the Seventh Conference*, volume 1610 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 1999.

38. R. Koschke. Survey on software visualization for software maintenance, re-engineering, and reverse engineering.
www.informatik.uni-stuttgart.de/ifi/ps/rainer/softviz.
39. Rainer Koschke. Software Visualization for Reverse Engineering. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
40. D. Lütke-Hüttmann. Knickminimales zeichnen 4-planarer clustergraphen. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, 1999.
41. J. Marks, editor. *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
42. P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM Journal on Optimization*, 11(4):1065–1080, 2001.
43. P. Mutzel, M. Jünger, and S. Leipert, editors. *Graph Drawing 2001*, volume 2265 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
44. P. Mutzel and R. Weiskircher. Bend minimization in orthogonal drawings via integer programming, 2001. submitted.
45. S. North and G. Woodhull. On-line hierarchical graph drawing. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 2002.
46. H. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.
47. A. Quigley and P. Eades. Fade: Graph drawing, clustering, and visual abstraction. In J. Marks, editor, *Graph Drawing (Proc. 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 197–210. Springer-Verlag, 2001.
48. E. M. Reingold and Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.
49. K. Ryall. Glide. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 476–477. Springer-Verlag, 2002.
50. K. Sugiyama, editor. *Graph drawing and applications for software and knowledge engineers.* World Scientific, 2002. to appear
51. K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. Softw. Eng.*, 21(4):876–892, 1991.
52. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.
53. R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
54. C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. In J. Marks, editor, *Graph Drawing (Proc. 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2001.
55. A. Winter. Exchanging graphs with GXL. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing (Proc. 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 482–496. Springer-Verlag, 2002.
56. Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*

On the Visualization of Java Programs

Holger Eichelberger and J. Wolff von Gudenberg

University of Würzburg
Am Hubland, 97074 Würzburg, Germany
{eichelberger,wolff}@informatik.uni-wuerzburg.de

Abstract.

In this paper we present a graph drawing framework that can be used to automatically draw UML class diagrams and a compiler that extracts the needed information from Java source code. Both components can be combined to a visualization tool for Java programs.

1 Introduction

UML class diagrams are frequently used to illustrate the static structure of an object-oriented program. A textual representation of these diagrams is usually kept in a repository either in XMI (XML Metadata Interchange)[17] format or in other textual representations like UMLscript [2]. The visualization of a software description given in such a format is a graph drawing problem that may be solved by applying well-known algorithms. An evaluation of current implementations of commercial tools showed, that usually unsuitable algorithms are used. Hence, we conclude that a UML class diagram cannot be drawn by simply applying well-known graph theoretic methods and that the problem of an appropriate drawing of a UML class diagram with all its sophisticated features is not solved satisfactorily so far.
We designed an algorithm which calculates a hybrid layout. The algorithm works on a dynamically classified pseudo-hierarchy and respects association classes, nested packages, inner classes and incremental drawing, even if not all features may be retrieved from source code by a specific Java parser like the one described in section 4.1. An implementation of this algorithm is useful for students visualizing their own projects, for teachers showing several problems of software engineering as well as the problems of applied graph drawing and it may be seen as a reference implementation for UML tools.
In this paper we describe our algorithm implemented in the graph drawing framework *SugiBib*, a pure Java framework. *SugiBib* reads its input from a UMLscript [2] specification. The implementation follows the toolbox principle so the steps described in subsection 3.2, which are implemented as highly configurable components, can be reused in other graph drawing algorithms.

2 Related Work

Our algorithm is developed from Seemanns extension [8] of the well-known Sugiyama-Algorithm [10]. Our implementation accepts UMLscript [2] as input language. Further possiblities to be realized in future are GXL (Graph Exchange Language) [13] and XMI (XML Metadata Interchange)[17].
Other approaches to the layout of UML class diagrams are presented in [6] and [12]. Unfortunately, these works emphasize the drawing a nice diagram according to the aestetic aspects of the graph drawing community (cf. [7]) intead of regarding the implicit semantics of software development diagrams.
The functionality of parsing the source code and displaying the static structure of a program is usually done by a CASE tool. In [1] we evaluated the layout facilities of 42 tools which promise UML conformity. The test which required sophisticated UML facilities according to UML version 1.3 showed that the layout capabilities of all regarded tools are disappointing.

3 SugiBib

SugiBib is a pure Java framework which has been developed to implement a generalization and extension of Sugiyama's algorithm [10]. The algorithm has been extended to draw graphs with 2 kinds of edges. Furthermore nodes in input graphs may occupy a two-dimensional, rectangular area and contain nested nodes, i.e. nodes representing a subgraph.
Such graphs correspond to UML class diagrams quite well, hence a specific instantiation of the framework for UML class diagrams is performed based on a modified and extended version of the Seemann algorithm [8]. The framework architecture has been chosen in order to test several alternatives for sub-algorithms.
Other approaches to calculate the layout of a class diagram usually rely on other well-known graph drawing algorithms like spring embedder [4] or orthogonalization [6]. A hierarchical approach (or at least hierarchical constraints) seem to be appropriate since software design documents are usually organized hierarchically.

3.1 Layout of UML Class Diagrams

It is obvious that classes map to nodes and associations as well as inheritance relations map to edges, the latter building the hierarchy. For class diagrams which do not contain a proper generalization hierarchy an aesthetic layout cannot be calculated without further consideration. A general partition of the set of edges into *hierarchical edges* and *non-hierarchical edges* has to be done. The set of hierarchical edges is seen as a pseudo hierarchy which can be laid out by the Sugiyama algorithm. The current version of SugiBib derives a pseudo hierarchy which may contain the inheritance hierarchy, the aggregation/composition edges, the associations or the dependencies by applying different heuristics.
Further steps are designed in order to realize incremental sucessive layout features, notes, *nested structures* and association classes.

Nested structures can be represented by a certain kind of edges which is seen as a part of the hierarchy. In some subalgorithms like calculation of ranks this is the appropriate way, but in others like calculation of coordinates these edges are not regarded. Therefore, we define the containment of nodes as a *second level hierarchy* which exists outside the set of edges of a graph. Edges of this hierarchy can be taken over into and removed dynamically from the primary hierarchy which determines the basic layout of the graph.
To achieve the layout of nodes which have to keep in a close vicinity, we introduce *compound nodes* in order to apply standard algorithms like rank calculation or crossing minimization. To keep the vicinity of `B and C` in figure 1, these nodes are inserted into the compound node `E`. All edges to `B and C` are mapped to `E` and `E` replaces `B` and `C`. The compound node is broken,in order to calculate the coordinates of the individual nodes, and the edges are mapped back to `B` and `C`.

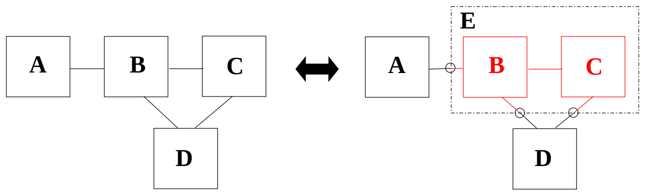


Fig. 1. Transformation of individual nodes to compound nodes and v.v.

To improve the aesthetic quality of UML diagrams several instantiations of the framework together with additional steps have been implemented.
The following choices describe the current instantiation of the framework for the visualization of Java programs.

- Choose the inheritance subgraph as hierarchy.
- Introduce nested nodes for inner classes or packages.
- Create compound nodes for specific comments.
- Use N-level backtracking (c.f. section 3.4) for the minimization of edge crossings. Usual heuristics like barycentric or median ordering have been tested with median success.
- Apply Sugiyama's original placement and Gansner's modification [5] and combine the result with an orthogonal layout of non-hierarchical edges.
- Keep compound nodes closely together, due to readability of the UML diagram.

3.2 The Algorithm

The following steps describe our approach to calculate the layout of a class diagram. In parallel we show the results of the several steps applied to an example in figure 2. Nodes which are contained in the package `sub` are given by their fully qualified name, edges (except of containment edges) are drawn like specified by the UML. Invisible nodes are shown by a crossed rectangle. In the input diagram in figure 2 a) a sequence of nodes in an arbitrary order is given.

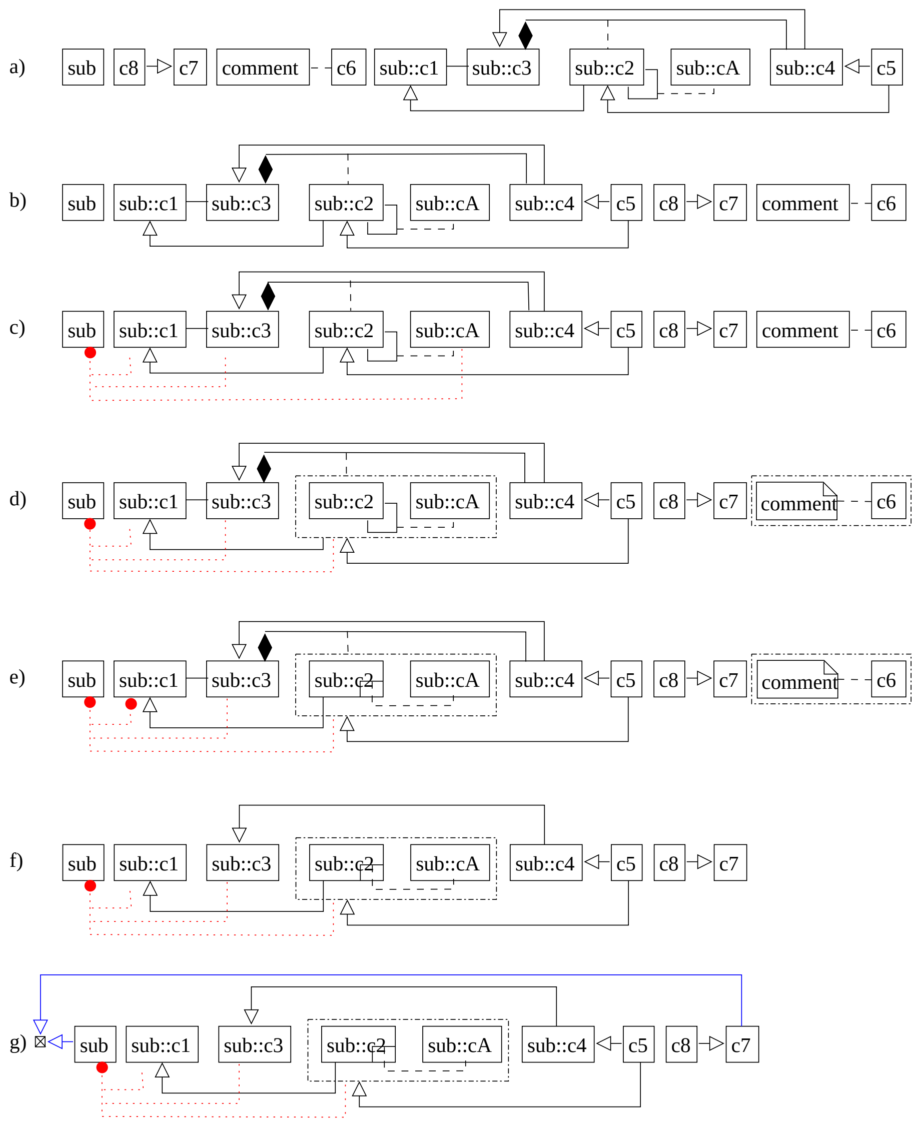


Fig. 2. The several steps of our algorithm applied to a diagram with association classes, a package and a comment note (continued in figure 3).

1. Identify the edges of the hierarchy (inheritance, aggregation, freely defined hierarchy).
2. Order the set of nodes with respect to nested nodes. Within a subsequence like a nested node order the top level nodes according to the following weights $weight(n) = \sum_{e \in edges(n)} \begin{cases} w_h & , if\ isHierarchical(e) \\ w_o & , else \end{cases}$ where $w_h > w_o$ in order to release the dependency between predefined sequences in the input graph and the result. This is shown in figure 2 b).
3. Insert the edges of the second-level hierarchy where no hierarchical edges are present so far. In figure 2 the containment hierarchy is shown as dotted edges from a node with a filled circle to the contained node. Figure 2 c) shows the graph after inserting the containment hierarchy.
4. Given some information about an intended neighborhood of nodes, create compound nodes. In figure 2 d) compound nodes which group `c6` and its comment as well as `sub::c2` and its association class `sub::cA` are inserted.
5. Remove loops by inserting them into the node, which is furthermore responsible for the correct layout of its reflective edges. This is drawn in figure 2 e) as a kind of inner edge in `sub::c2`.
6. Remove non-hierarchical edges and disconnected nodes from the graph temporarily in order to apply the Sugiyama algorithm. Figure 2 f) shows this.
7. Insert a virtual root, if the graph consists of more than one component. The root of a graph may be used as starting point of some subalgorithms. The edges which connect the virtual root to the other nodes shown in figure 2 g) are depicted as hierarchical edges, even if these edges are not visible in the final drawing.
8. Calculate the ranking of the graph. The result is depicted in figure 3 h).
9. Reduce the number of crossings by reordering the nodes of each rank.
10. The non-hierarchical edges which have been removed temporarily in step 6 are reinserted. In figure 3 i) the association between `sub::c1` and `sub::c3` and the composition and the connection to its association class are resinserted.
11. Remove the second level hierarchy because the information is not considered in the following steps. This is shown in figure 3 j).
12. Perform subtree crossing optimizations in order to rearrange the graph and eliminate edge crossings which have to be expected by regarding the non-hierarchical subset of edges which was integrated in the last step.
13. Iteratively calculate the coordinates of the nodes with respect to the space needed by nodes and edges connected to the nodes. Especially arrange the contained nodes in a close vicinity and calculate packages e.g. in the same step. Then calculate the layout of the non-hierarchical edges. The default layout for non-hierarchical edges is an orthogonal layout as shown in figure 3 k). Note that association classes are not respected in this step.
14. Reintegrate compound nodes which have been inserted in step 4 except of disconnected nodes removed in step 6. Because coordinates have been assigned to compound nodes in the last step, the graph has to be laid out again

by searching minimum displacements for the affected nodes. This is shown in figure 3 l).

15. Search an optimal placement for disconnected nodes and reinsert them into the graph. The final layout including the hidden edges to the virtual root is shown in figure 3 m).

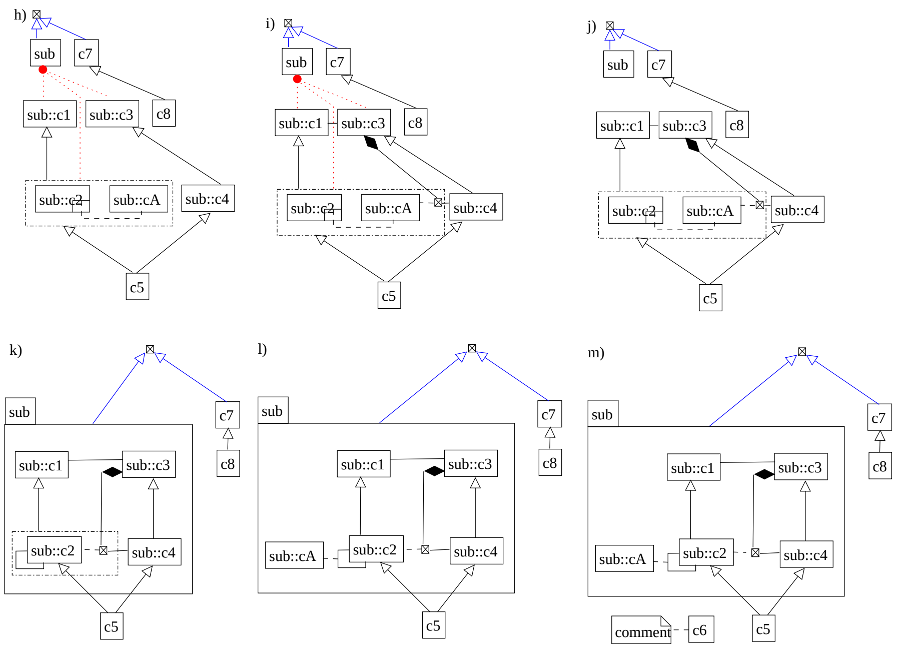


Fig. 3. The several steps of our algorithm (continued).

3.3 The Framework

The framework for the drawing of graphs as described above may be instantiated and adapted in 2 ways. Choose the appropriate algorithm for the particular application or insert more steps.
Sugiyama's original algorithm (steps 8,9,13) layouts an acyclic digraph as a leveled structure. A transformation to an acyclic graph may proceed the computation, if necessary.

3.4 Minimization of Edge Crossings

N-level backtracking is a newly developed edge-crossing minimization algorithm. The usual heuristics (except of [11] e.g.) used for edge-crossing minimization are clearly superior in run-time but they only look into adjacent ranks.

Algorithm 1 N-level backtracking crossing minimization

```
Require: $G^R = (V^R, E)$ as input, ranks are assigned to all nodes
  for all $v \in V^R$ do
    $unmark(v)$
  end for
  repeat
    $c_{last} = crossings(G^R)$
    for $r = 0$ to $\max_{v \in V^R} rank(v)$ do
      $nodes(r) = \{v | v \in V^R \wedge rank(v) = r\}$
      if $|nodes(r)| > 1$ then
        $regardRank(G^R, nodes(r))$
      end if
    end for
  until $|\{v | v \in V^R \wedge \neg marked(v)\}| = 0 \vee c_{last} < crossings(G^R)$
```

Algorithm 2 regardRank

```
Require: $G^R = (V^R, E), N \subseteq V^R$ as input
  $n = node$ with $\max_{n_0 \in \{n_1 | n_1 \in N \wedge \neg marked(n_1)\}} |children(n_0)|$
  for all $m \in N - n$ do
    $c_0 = rankcrossings(r)$
    $c_G = crossings(G^R)$
    $exchange(n, m)$
    if $c_0 > rankcrossings(r)$ then
      $regardRank(G^R, children(m))$
    end if
    if $c_G > crossings(G^R)$ then
      $mark(m)$
    else
      $exchange(m, n)$
      $exit\ loop$
    end if
  end for
```

4 javac2UMLscript

4.1 JTransform

Information on source code to be analyzed may be retrieved from the parsers implemented in source code engineering tools like SNiFF+ or CASE tools like Rational Rose or MID Innovator. Because most parsers are not implemented as a platform independent API (Application Programming Interface) which may be distributed free from vendor licences we decided to implement a common parsing API for source code in Java. The parsing API called *JTransform* implements an object-oriented parser which works on the instructions and method calls included in the source code. *JTransform* creates a configurable parse tree which represents pieces of comments as well as dependencies given by package imports, nested and inner classes, inheritance relations and structural features of classes. Different kinds of iterators on the parse tree produce different kind of output. By subclassing the API it can be specialized to analyze code as well as comments. The following list gives some of the possible applications of *JTransform*:

1. Transform source code into XMI (XML Metadata Interchange), UMLscript [2] (the program is called javac2UMLscript), GXL [13]. Check the source for backward conformance to the model when the source changes.
2. Calculate simple source code and object-oriented metrics.
3. Work on formatted comments, like specified for the javadoc tool:
   a) Check the comment of a method against the signature of the methods.
   b) Generate javadoc templates from signatures into the source code.
   c) Transform comments written in HTML into XML, XHTML or other formats.
4. Check or provide pretty printing of source code and comments.

In order to visualize Java software especially the transformation of source code into meta formats like XMI and UMLscript is important. But also the backward direction if a piece of source code is conformant to a meta language specification of the software system is interesting.

4.2 Detailed Features

Our compiler javac2UMLscript translates Java source code to UMLscript. It is generated as an extendable parser by the parser generator JavaCup [14] constructing a parse tree and visitor interface for this tree. The UMLscript generator is one possible visitor. It is able to generate more semantic information than a listing of classes with their attributes and methods. Inner classes, generalizations and associations are recognized. Information about imported classes is gathered. Methods are classified into different categories by their intended purpose. There are predefined categories like ≪constructor≫, categories which can be specialized by the user by denoting the entries as regular expressions (like in PERL) in the project file and user defined categories.

The input to the compiler is a project file containing the source codes to be parsed and some constraints controling the visualization process.

5 Conclusions and Further Development

In [1] we described, that current tools dealing with UML are not capable to calculate an appropriate layout of UML class diagrams. Therefore we proposed an algorithm which respects most of the sophisticated features of UML class diagrams like association classes, containment of nodes in packages and attached comments. Furthermore we described an extendible parser which can be used to test new approaches in the recognition of design patterns and the visualization of differences between several versions of a program.

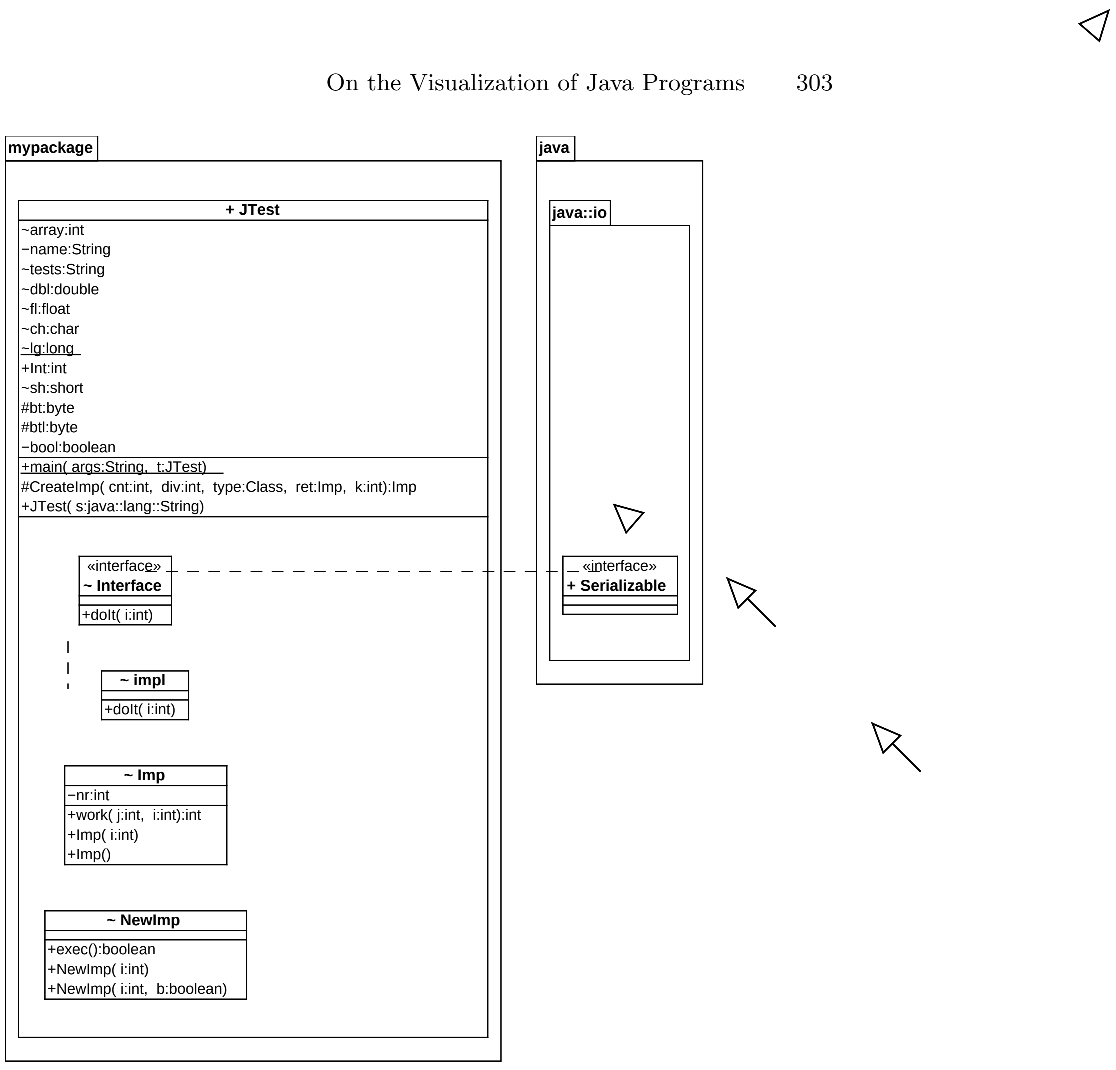


Fig. 4. A Java compiler test file which contains a lot of structural features like packages and a non-standard notation of inner classes (layout of nested structures).

SugiBib can be used in different ways: as a standalone class diagram viewer, as a browser component, a batch rendering application and as a layout component to be plugged into a common CASE tool. A possible drawback may be the input language UMLscript. Hence, we will consider to switch the input language. This will considerably enlarge the applicability of the framework.

The parser will be extended by more visitors that extract information on design patterns, mark differences between two versions of a program, or work on formatted comments (cf. 4.1).

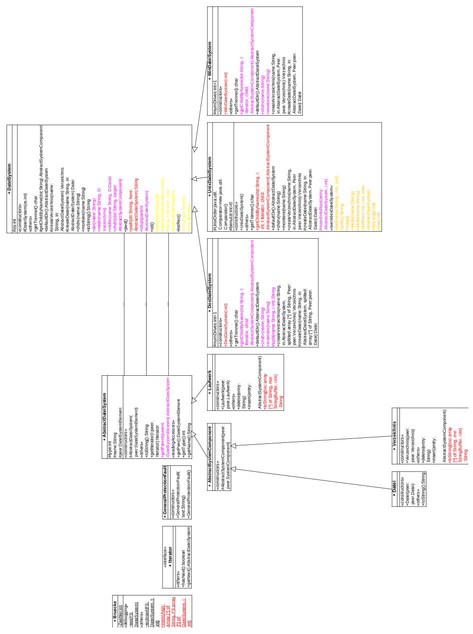


Fig. 5. A diagram generated by SugiBib showing a modified bridge pattern. Placement of isolated classes will be improved.

6 Sample Pictures

Some sample pictures demonstrate the current state of the implementation. They are automatically generated from Java source code which is available on request. Different colors (or gray shadings in this printout) indicate further features of

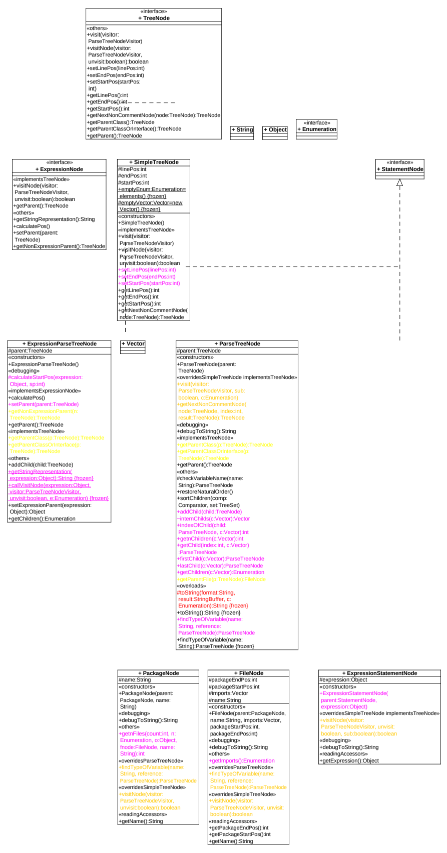


Fig. 6. The core diagram of JTransform. Most of the parse tree nodes have been omitted. Ellipses (...) will be generated by a future version of javac2UMLscript.

methods which are also obtained by source code analysis. Currently the orthogonal layout of flat edges is not optimal, since their relative positions of hidden nodes are determined too early and the implementation of the layout of association classes is not finished so far.

References

1. H. Eichelberger: *Evaluation of the Layout Facilities of CASE Tools*, internal report, available on request
2. H. Eichelberger, J. Wolff von Gudenberg: *UMLscript language specification* (in German), Technical Report No. 272, University of Wuerzburg, February 2001
3. H. Eichelberger, J. Wolff von Gudenberg: *JTransform - A Java Source Code Transformation API*, Technical Report, University of Wuerzburg, to appear, 2001
4. T.M.J. Fruchterman, E. M. Reingold: *Graph Drawing by Force-directed Placement* Software - Practice and Experiences, 21(11):1129-1164, November 1991
5. E.R. Gansner, E. Koutsofious, S.C. North, K.-P. Vo: *A Technique for Drawing Directed Graphs*, IEEE Transactions on Software Engineering, SE-19(3):214-230, March 1993
6. C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, P. Mutzel: *Automatic Layout of UML Class Diagrams*, in Mathematics and Visualization, P. Mutzel (Editor): Graph Drawing - 9th Internation Symposium, Springer, 2001
7. H.C. Purchase, J-A. Allder, D. Carrington: *User Preference of Graph Layout Aestetics: A UML Study*, in LNCS 1984: J. Marks (Editor): Graph Drawing - 8th Internation Symposium, p. 5-18, Springer, 2000
8. J. Seemann: *Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams*, Lecture Notes in Computer Science, LNCS 1353 G. DiBattista (Editor), 414-423, 1998
9. J. Seemann, J. Wolff von Gudenberg: *Attributierte Graphen zur Metamodellierung mit UML*, UML-Erweiterungen und Konzepte der Metamodellierung, April 2000, via `http://IST.UniBw-Muenchen.DE/GROOM/META/Abstracts/Seemann.pdf`
10. K. Sugiyama, S. Tagawa, M. Toda: *Methods for Visual Understanding of Hierarchical System Structures*, IEEE Transactions on Systems, Man, and Cybernetics, SMC-11(2):109-125, February 1981
11. K. Sugiyama, K. Misue: *Visualization of Structural Information: Automatic Drawing of Compound Digraphs*, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21(4):876-891, July/August 1991
12. R. Wiese, M. Eigelsperger, M. Kaufmann: *yFiles: Visualization and Automatic Layout of Graphs*, in Mathematics and Visualization, P. Mutzel (Editor): Graph Drawing - 9th Internation Symposium, Springer, 2001
13. A. Winter, B. Kullbach and V. Riediger: *An Overview of the GXL Graph Exchange Language* in Proceedings of Dagstuhl Seminar on Software Visualization, 2001
14. *CUP Parser Generator for Java*, homepage of the CUP: `http://www.cs.princeton.edu/~appel/modern/java/CUP/`
15. *Layout in UML and CASE*, homepage of SugiBib: `http://www.sugibib.de/english.html`
16. *OMG Unified Modeling Language Specification*, Version 1.4, February 2001 via `http://www.omg.org`
17. XML Metadata Interchange (XMI) Version 1.1, via `http://cgi.omg.org/docs/ad/99-10-02.pdf`

Graph Drawing Algorithm Engineering with AGD

Carsten Gutwenger[1], Michael Jünger[2], Gunnar W. Klau[3], Sebastian Leipert[1], and Petra Mutzel[3]*

[1] caesar Foundation, Bonn, Germany
[2] University of Cologne, Germany
[3] Vienna University of Technology, Austria

Abstract.

We discuss the algorithm engineering aspects of AGD, a software library of algorithms for graph drawing. AGD represents algorithms as classes that provide one or more methods for calling the algorithm. There is a common base class, also called the type of an algorithm, for algorithms providing basically the same functionality. This enables us to exchange components and experiment with various algorithms and implementations of the same type. We give examples for algorithm engineering with AGD for drawing general non-hierarchical graphs and hierarchical graphs.

1 Introduction

One of the most important points of criticism that designers of graph drawing software must face is that drawing styles are fixed and their is usually little freedom for the users to influence the results according to their specific needs. On the other hand, the most popular paradigms for drawing hierarchical and non-hierarchical graphs are really only methodological frames that require specific decisions to be made by the software designer. E.g., Sugiyama-style methods consist of three phases: layering, crossing minimization, coordinate assignment, and for each phase there is a variety of possible implementations to choose from. Similarly, the planarization method is really a frame that consists of the three phases "topology-shape-metrics", in the first of which the drawing's topology is fixed, in the second, the shape of the edges is determined, and in the third, vertex and edge bend coordinates are assigned. Again, there is a variety of implementations to choose from, and these decisions have a great impact on the appearance of the final drawing.

* Corresponding author: Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstr. 9-11 E186, A-1040 Wien, Austria. Email: mutzel@ ads.tuwien.ac.at. This work is partially supported by DFG-Grant Mu 1129/3-2 and Ju 204/7-4 Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen".

The development of the AGD software, an object-oriented C++ class library of **A**lgorithms for **G**raph **D**rawing, has started in 1996. The goal has been to provide users with a toolbox to create their own implementations of graph drawing algorithms according to their specific needs and thus, bridge the gap between theory and practice in the area of graph drawing.

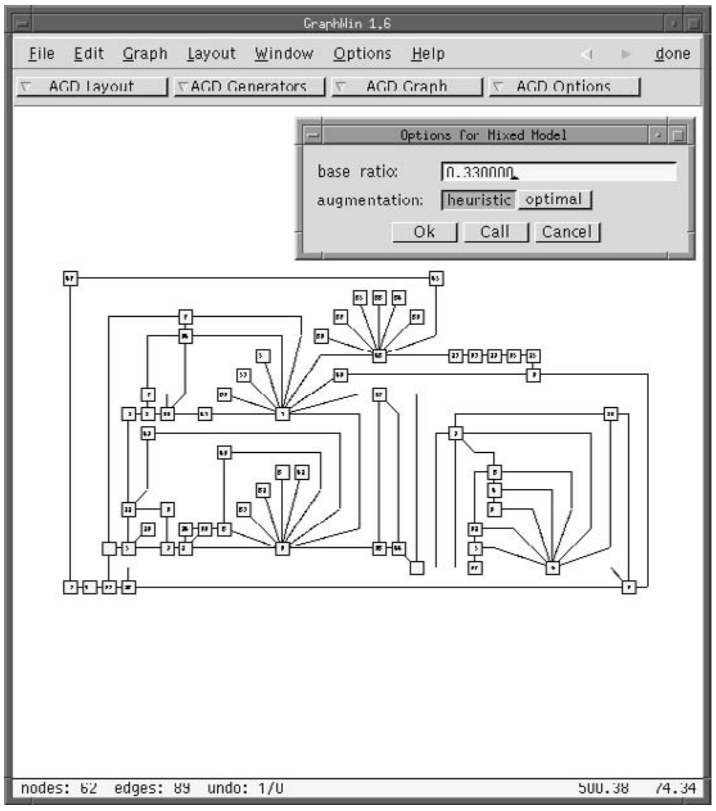
Algorithm Engineering aims at providing new techniques and tools for the design of efficient algorithms and for the formal and experimental analysis of the performance of algorithms.

For drawing general non-hierarchical graphs, Batini *et al.* [BTT84,BNT86] suggested a method based on planarization which often leads to good drawings for many applications. However, until 1996, no publically available software layout tool used the planarization method. The reason for this was twofold: On the one hand, a lot of expertise is necessary concerning planarity testing algorithms, combinatorial embeddings, planar graph drawing algorithms, and (often NP-hard) combinatorial optimization problems. On the other hand, great effort is needed to implement all necessary algorithms and data structures, since the planarization method consists of various phases that require complex algorithms.

Recently, major improvements have been made concerning the use of the planarization method in practice (e.g., [JM96b,FK96,GM98,GM00,GMW01]). Today, there exist some (publically available) software libraries using the planarization method successfully for practical graph layout [AGD00,GDT,GT98]. In AGD, the planarization method is implemented in a modular form, so that it is easy to experiment with different approaches to the various subtasks of the whole approach. This enables experimental comparisons between various algorithms in order to study and understand their impact on the final drawing. Not only in graph drawing, the empirical study of combinatorial algorithms is getting increasing attention.

Another reason for building AGD was our intention to show how mathematical methods can help to produce good layouts. Many of the optimization problems in graph drawing are NP-hard. However, this does not mean that it is impossible to solve them in practice. AGD shows that problem instances can often be solved to provable optimality within short computation time by using polyhedral combinatorics and branch-and-cut algorithms.

The most important design feature in AGD for algorithm engineering is the representation of algorithms as classes that provide one or more methods for calling the algorithm. Thus, a particular instance of an algorithm is an object of that class, which can also maintain optional parameters of the algorithm as member variables. Algorithms providing basically the same functionality (e.g., computing a subgraph or drawing a graph) are derived from a common base class, which we call the *type* of the algorithm, i.e., algorithms of the same type support a common call interface. This allows to write generic functions that only know the type of an algorithm. The type is rather general, but can be refined by declaring a *precondition* (e.g., the input graph has to be biconnected or planar) and a *postcondition* (e.g., the produced drawing is straight-line and contains no crossings). The precondition specifies how the algorithm can be applied safely.



base ratio: 0.330000
augmentation: [✓ heuristic] optimal

```
nodes: 62   edges: 89   undo: 1/0                    500.38    74.34
```

Fig. 1. A screenshot of AGD showing a mixed-model drawing and its options

We call an instance of an algorithm together with its pre- and postcondition a *module*. Pre- and postconditions are sets of basic properties (e.g., properties of graphs like planar, acyclic or biconnected, or properties of drawings like orthogonal or straight-line). AGD maintains dependencies between these properties, such as "biconnected implies connected", or "a tree is a connected forest", in a global rule system.

AGD provides a general concept for modeling subtasks of algorithms as exchangeable modules. Such a subtask is represented by a *module option* that knows the module type, a guaranteed precondition (which always holds when the algorithm is called), and a required postcondition (which must hold for the output of the algorithm). The current module itself is stored as a pointer. In order to set a module option, a particular module is passed and automatically checked if it satisfies the requirements, i.e., it has the correct type, the guaranteed precondition implies its precondition, and its postcondition implies the required postcondition. These implications are checked using the global rule system for properties. Figure 1 shows a screenshot of AGD displaying a graph with 62 vertices and 89 edges drawn with a planar drawing algorithm.

In this paper, we would like to demonstrate how the design of AGD supports algorithm engineering for graph drawing. We can group the graph drawing algorithms contained in AGD according to the classes of graphs for which they are useful. Section 2 describes algorithm engineering in the context of drawing general non-hierarchical graphs. We focus on the planarization method, for which algorithm engineering is especially interesting in the pursuit of finding most pleasing drawings by experimenting with various approaches to the generic basic phases of the method. Also hierarchical graph drawing consists of various phases that are interesting subjects for an algorithm engineering approach. Section 3 discusses the modular design in AGD for drawing hierarchical graphs. Finally, some technical details of the design of AGD are given in Section 4.

2 Algorithm Engineering for Non-hierarchical Graphs

In this section we will mainly focus on the planarization method and its implementation in AGD. General non-hierarchical graphs can also be drawn using force-directed methods. Indeed, most available software tools for graph drawing use force-directed methods. They are especially useful for drawing very sparse, tree-like graphs. AGD contains implementations of the spring-embedder algorithm by Fruchterman and Reingold [FR91], and the algorithm by Tutte [Tut63]. However, for many applications, e.g., data base visualization or software design, the planarization method leads to much nicer layouts.

The idea of the *planarization method* is to first transform the given graph into a planar graph and then apply a planar graph drawing algorithm for creating the layout. The method draws on many beautiful results that graph theorists and algorithm designers have obtained in the last few decades. It typically produces drawings with a small number of crossings. AGD contains a very flexible implementation of the planarization method (`PlanarizationGridLayout`) that was originally proposed in [BTT84,BNT86]. So far, all planar layout routines in AGD generate grid drawings, i.e., the computed coordinates are integer. This is supported with the `GridLayoutModule`. The AGD modules involved in the planarization method are shown in Fig. 2.

2.1 The Planarization Phase

In the planarization phase, realized in AGD by the class `PlanarizerModule`, any procedure transforming the given graph G into a planar graph G' is allowed. One can imagine that a method generating a random or a force-directed layout and then substituting the crossings by artificial vertices could be a `PlanarizerModule`. AGD offers the possibility of adding and creating any new

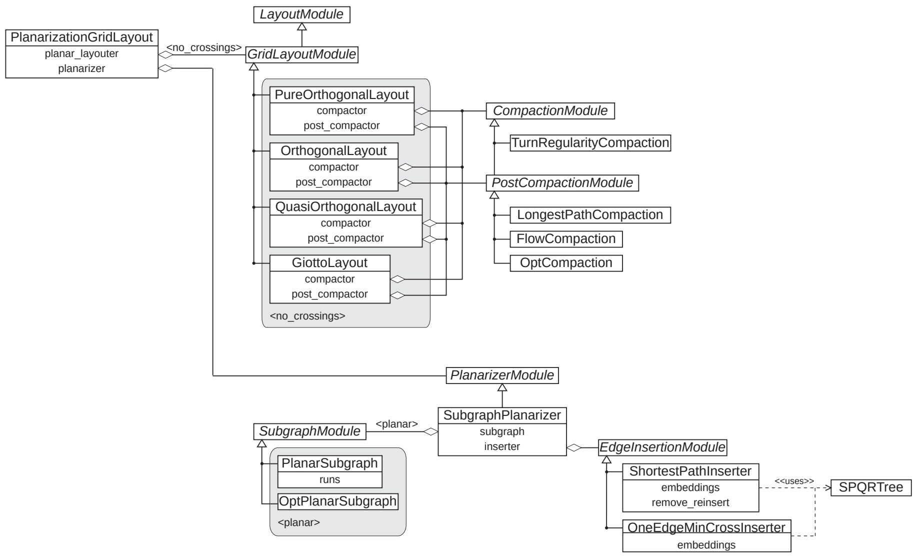


Fig. 2. Modules for drawing non-hierarchical graphs

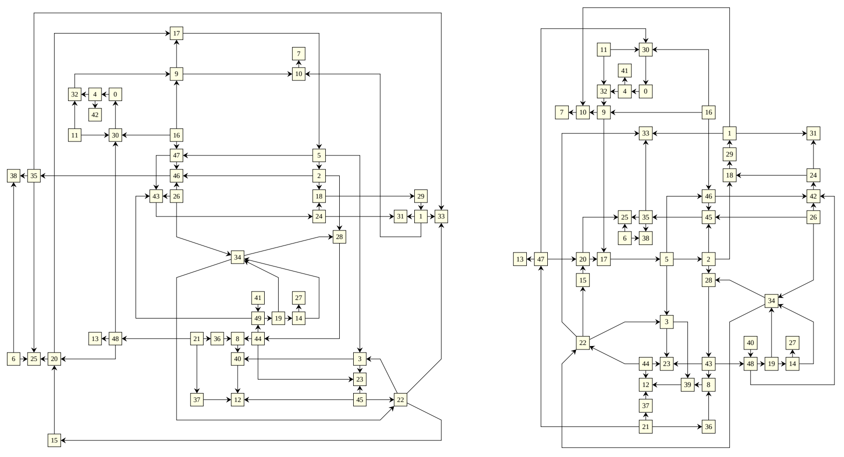
method for planarizing a graph via substituting edge crossings by artificial vertices.

Our experiments have shown that, in order to keep the number of crossings small, it is advantageous to use a `SubgraphPlanarizer` module. Subgraph planarization works in two steps: First, a (large) planar subgraph of the input graph G is computed. Then, the deleted set of edges is reinserted into the planar graph while trying to keep the number of edge crossings small. In order to get the transformed planar(ized) graph G', the crossings are substituted by artificial vertices.

The problem of finding a planar subgraph of maximum size is NP-hard. AGD contains a heuristic `PlanarSubgraph` [JLM98] based on PQ-trees, which achieves good results in practice, and a branch-and-cut algorithm `OptPlanarSubgraph` to solve the problem to optimality [JM96b]. Both algorithms are implemented as a `SubgraphModule`. This makes it easy to select an existing one or to add a new algorithm as a new subgraph module.

In the following, we will concentrate on the algorithm engineering aspects for the edge reinsertion phase. This step determines the number of crossings in the final drawing. When reinserting the removed edges, the edge crossings are represented by artificial vertices with degree four. Edge insertion algorithms are derived from the module type `EdgeInsertionModule` (see Fig. 2). The problem of reinserting all the edges such that the number of crossings is minimized is again NP-hard.

Heuristics often proceed iteratively and insert the edges one after the other. The standard procedure in the literature [EM99,TBB88] is to deter-



(a) Fixed embedding (b) Optimal embedding

Fig. 3. The influence of the combinatorial embedding

mine a combinatorial embedding and then to insert the edges iteratively while keeping the embedding fixed. This method is implemented in AGD as module `ShortestPathInserter`. However, fixing the embedding can lead to many more crossings than necessary even if only one edge needs to be reinserted.

Recently, Gutwenger, Mutzel, and Weiskircher [GMW01] have presented a conceptually simple linear-time algorithm (based on SPQR-trees) for inserting an edge into a planar graph with the minimum number of crossings over the set of all combinatorial embeddings. In AGD, this algorithm is implemented in the module `OneEdgeMinCrossInserter`.

All edge insertion modules provide optional heuristics which can improve the quality of the solution significantly. The two algorithms that fix an embedding allow calling the algorithm for several randomly generated embeddings and select the best solution. The number of embeddings is controlled by the parameter `embeddings`. Generating random combinatorial embeddings requires computing the SPQR-tree for each biconnected component [DT96]. AGD provides a linear-time implementation of SPQR trees [GM00] in the class `SPQRTree`.

Experiments show that significant improvements are achieved if, in a post-processing step, a set S of edges is removed from the graph and reinserted. In each such step, the number of crossings can only decrease. All three edge insertion algorithms support this heuristic, which is controlled by the parameter `remove_reinsert`. Possible settings are `none` (skip postprocessing), `inserted` (apply postprocessing with all reinserted edges), and `all` (apply postprocessing with all edges in the graph).

Figure 3(a) shows a graph for which the standard iterative edge insertion leads to 14 crossings if a random combinatorial embedding is chosen. In this case the drawing has a grid size of 22×22. However, when taking the optimum one edge insertion module and adding the remove and reinsert algorithm for all edges, the resulting drawing has only 11 crossings and has size 16×22 (see Fig. 3(b)). This is a good example that shows how the size of the drawing increases with the number of crossings.

2.2 The Planar Graph Drawing Phase

The planarization phase leads to a planar graph G' containing artificial vertices. Replacing all artificial vertices with edge crossings in a drawing of G' results in a drawing of the input graph G. We prefer to use orthogonal or quasi-orthogonal algorithms because, in this case, an edge crossing is drawn as two crossing horizontal and vertical line segments. In orthogonal graph drawing (all edge segments are drawn horizontally or vertically), an important goal is to keep the number of bends small.

Many minimum cost flow-based algorithms are available for this task (see Fig. 2): In a first step (bend minimization) a flow in an underlying network determines the shape of the orthogonal drawing. A second step (compaction) deals with assigning the coordinates to the vertices and bends. The library contains the classical bend-minimizing algorithm of Tamassia [Tam87], which is applicable if the maximum degree of the input graph is at most four (`PureOrthogonalLayout`). For graphs with vertices of higher degree, AGD contains three extensions of Tamassia's algorithm: Giotto (`GiottoLayout`) [TBB88] and two variations of the quasi-orthogonal drawing algorithm in [KM98] (`QuasiOrtogonalLayout` and `OrthogonalLayout`).

The AGD library contains several algorithms for the compaction phase within the topology-shape-metrics approach. Construction heuristics assign coordinates to vertices and bends of a given orthogonal representation which encodes the shape of a planar orthogonal drawing. Improvement heuristics operate directly on a layout and try to decrease its total edge length and area. This division is reflected in the library: The user chooses a construction method (from `CompactionModule` or `PostCompactionModule`) and optional improvement heuristics (from `PostCompactionModule`). The former transforms the orthogonal representation by introducing artificial edges and vertices. By changing the options of the compaction algorithms, there are different techniques available for this transformation. AGD's constructive heuristics include longest path-based or flow-based compaction with rectangular dissection (`LongestPathCompaction` and `FlowCompaction`) [Tam87] and two variants of a flow-based compaction technique based on the property of turn-regularity (`TurnRegularityCompaction`) [BDD$^+$00]. For the improvement phase, AGD offers iterative application of compaction with longest path or flow computations (`LongestPathCompaction` and `FlowCompaction`) as used in the area of VLSI-design, see, e.g., [Len90]. In addition, AGD provides an implementation of the integer linear programming-based approach by Klau and Mutzel (`OptCompaction`) [KM99] that produces an optimum drawing in terms of minimum total length or maximum edge length.

The modular design of the compaction phase proved very useful in a recent experimental study [KKM01]. All combinations of constructive and improvement

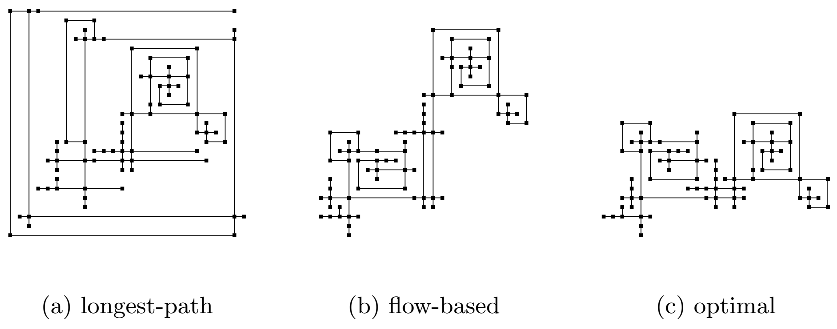


Fig. 4. The influence of different compaction algorithms

heuristics could easily be compared against each other and against the optimum values provided by the integer linear programming-based algorithm. One of the main insights of this study has been that flow-based compaction should always be used as an improvement method. Figure 4 shows the output of two different compaction strategies and a corresponding optimum solution.

2.3 Planar Graphs

In addition to the (quasi-)orthogonal planar drawing algorithms described with the planarization method, the following planar drawing algorithms are contained in AGD (the corresponding AGD modules are shown in Fig. 5).

AGD contains an implementation of the Kandinsky algorithm `KandinskyLayout`) [FK96] which is not yet available as a module in the planarization method, but can be used for planar graphs with arbitrary vertex degrees. Unlike other flow-based orthogonal drawing methods, the Kandinsky algorithm places vertices and bends as points on a coarse grid and routes the edges in a finer grid as sequences of horizontal and vertical line segments. A variant of the algorithm uses a common grid for vertices and edges. Vertices are represented as boxes whose size is bounded by the vertex degree.

Probably, the best known planar graph drawing algorithm is the one by de Fraysseix et al. [DPP90]. This seminal paper shows that a planar graph with n vertices can always be drawn without bends and crossings on a grid of size polynomially bounded in n. The idea is to first augment the graph by additional edges in order to obtain a triangulated planar graph. Then, a so-called canonical ordering for triangulated planar graphs is computed, and finally, the vertices are placed iteratively according to this ordering. Theoretically, the straight-line planar drawing problem was solved. However, the drawings did not look nice, especially not after the deletion of edges added in the augmentation step. Also, the angular resolution was not good. Recently, some work has been done to improve the aestetic quality of the drawings. Generalizing the canonical ordering

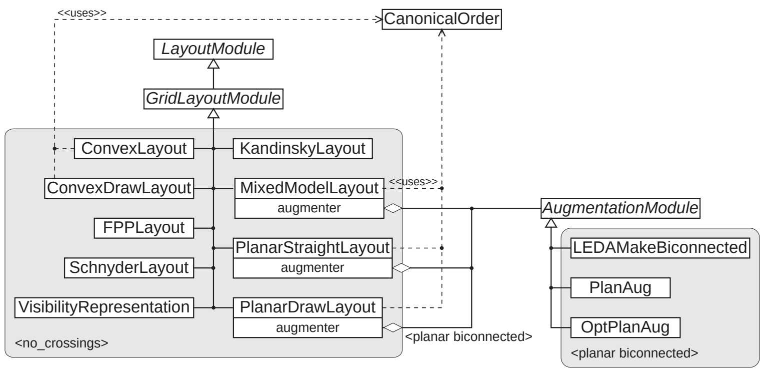


Fig. 5. Modules for drawing planar graphs

to triconnected [Kan96] and to biconnected planar graphs [GM97] already leads to a big improvement. This can be seen by using AGD, since implementations of all three canonical orderings (`CanonicalOrder`) and the corresponding placement algorithms are included in the system (`FPPLayout`, `ConvexLayout`, and `PlanarStraightLayout`).

The problem of the angular resolution has been solved by introducing some bends within the lines representing the edges, leading to pleasant polyline drawings [Kan96,GM98]. AGD contains an implementation of the mixed-model algorithm by Gutwenger and Mutzel [GM98] (`MixedModelLayout`). Figure 1 shows a screenshot of AGD displaying a graph drawn with the mixed-model algorithm.

In order to apply the drawing algorithms to planar graphs that are not necessarily biconnected, augmentation algorithms (realized as a `AugmentationModule`) are used for augmenting a planar graph to a biconnected planar graph. This augmentation problem consisting of adding the minimum number of edges is NP-hard. AGD provides a simple heuristic using depth-first-search (`LEDAMakeBiconnected`), the 5/3-approximation algorithm by Fialko and Mutzel [FM98] (`PlanAug`) that in most cases yields a solution which is very close to an optimum solution, and a branch-and-cut algorithm for exact optimization (`OptPlanAug`) [Mut95,JM94,Fia97].

In addition, AGD contains implementations of planar graph drawing algorithms, e.g., two algorithms for producing convex drawings of triconnected planar graphs (`ConvexLayout` [Kan96] and `ConvexDrawLayout` [CK97]), generalizations of these algorithms to general planar graphs [GM97] (`PlanarDrawLayout`), and an algorithm for producing weak visibility representations [RT86] (`VisibilityRepresentation`). Recently, we started to experiment with planar cluster drawing algorithms. Here, certain sets of vertices ("clusters") that may be nested, need to be placed within convex regions.

Some of the mentioned planar graph drawing algorithms yield very good results in terms of angular resolution, readability, area or number of bends (in

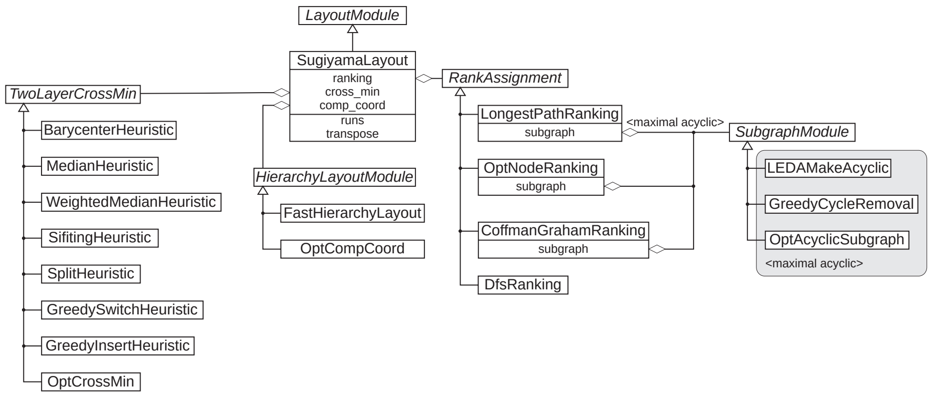


Fig. 6. Modules for drawing hierarchical graphs

particular, the Kandinsky and the mixed-model algorithm, also see Fig. 1). It is our long term goal to make these algorithms available in AGD within the planarization approach. The problem with these algorithms is that they cannot easily cope with the artificial (crossing) vertices of G'.

3 Algorithm Engineering for Hierarchical Graphs

Hierarchical graphs are directed graphs that must be drawn such that all edges run in the same direction (e.g., downward). Since this is only possible for acyclic graphs, non-acyclic graphs are made acyclic by temporarily reversing some edges. AGD provides a flexible implementation of the *Sugiyama algorithm* [STT81] (see Fig. 7 for two sample drawings), represented by the module `SugiyamaLayout` that consists of three phases. For each phase, various methods have been proposed in the literature. The available AGD modules and their dependencies are shown in Fig. 6.

In the first phase, handled by modules of type `RankAssignment`, the vertices of the input graph G are assigned to layers. If G is not acyclic, then we compute a maximal acyclic subgraph and reverse the edges not contained in the subgraph. The problem of finding an acyclic subgraph of maximum size is NP-hard. AGD contains two linear-time heuristics for solving the problem (`LEDAMakeAcyclic` based on depth-first-search and a greedy algorithm (`GreedyCycleRemoval`) [EL95]), as well as a branch-and-cut algorithm (`OptAcyclicSubgraph`) [GJR85] which is able to solve the problem to provable optimality within short computation time.

Currently, AGD contains the following algorithms for computing a layer assignment for an acyclic graph in which the edges are directed from vertices on a lower level to vertices on a higher level. `LongestPathRanking` is based on the computation of longest paths and tries to minimize the number of layers (height of

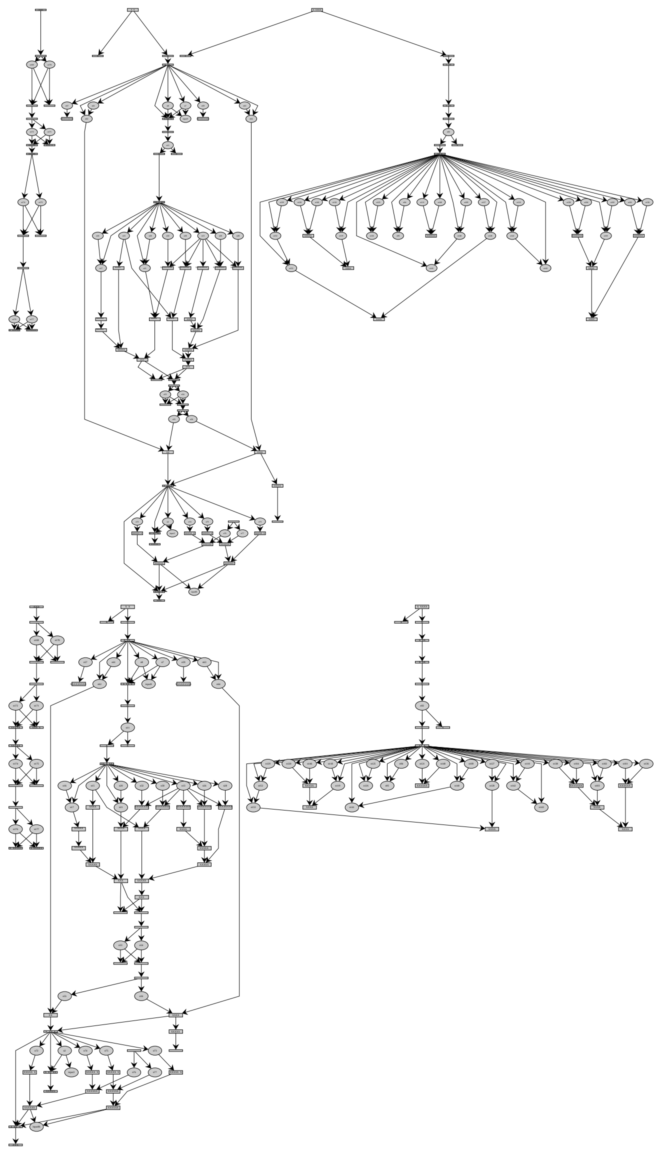


Fig. 7. A hierarchical graph drawn with different modules for the third phase

the drawing), `OptNodeRanking` minimizes the total edge length [GKNV93] (here the length of an edge is the number of layers it spawns), `CoffmanGrahamRanking` computes a layer assignment with a predefined maximum number of vertices on a layer (width of the drawing) [CG72], and `DfsRanking` simply uses depth-first-search and handles general graphs. If edges spawn several layers, they are split by inserting additional artificial vertices such that edges connect only vertices on neighboring layers.

The second phase determines permutations of the vertices on each layer such that the number of edge crossings is small. The corresponding optimization problem is NP-hard. A reasonable method consists of visiting the layers from bottom to top, fixing the order of the vertices on the layer and trying to find a permutation of the vertices on the upper next layer that minimizes the number of crossings between edges connecting the two adjacaent layers (2-layer crossing minimization). Then, the algorithm proceeds from top to bottom and so on until the total number of crossings does not decrease anymore. `SugiyamaLayout` contains a sophisticated implementation of this method, which uses further improvements like calling the crossing minimization several times (controlled by the parameter `runs`) with different starting permutations, or applying the `transpose` heuristic described in [GKNV93].

Several heuristics for 2-layer crossing minimization have been proposed. AGD provides implementations of the `BarycenterHeuristic` [STT81], `MedianHeuristic` [EW86], `WeightedMedianHeuristic` [GKNV93], `SiftingHeuristic` [MSM99], `SplitHeuristic` [EK86], `GreedySwitchHeuristic` [EK86], and `GreedyInsertHeuristic` [EK86]. Furthermore, a branch-and-cut algorithm for optimum solutions based on [JM96a] is implemented (`OptCrossMin`).

The third phase, handled by modules of type `HierarchyLayoutModule`, computes the final coordinates of the vertices and bend points of the edges, respecting the layer assignment and ordering of the vertices on each layer. AGD contains two implementations. The first (`OptCompCoord`) tries to let edges run as vertical as possible by solving a linear program, the second (`FastHierarchyLayout`) proposed by Buchheim, Jünger, and Leipert [BJL00] guarantees at most two bends per edge and draws the whole part between these bends vertically. Figure 7 shows a hierarchical graph drawn with the method proposed in [BJL00] (top), and drawn with the LP-based approach (bottom).

Some more specific classes of hierarchical graphs require algorithms that exploit their special structure. E.g., trees can be drawn nicely in AGD using the algorithm by Reingold and Tilford [RT81] and Walker [Wal90]. Moreover, st-planar graphs can be drawn by the algorithm suggested in [DTT92].

4 Design Details

AGD [AGD00] is an object-oriented C++ class library, which is based on the two libraries LEDA [MN99] and ABACUS [JT00]. LEDA provides basic data types and algorithms, e.g., the data type for the representation of graphs. ABACUS is a framework for the implementation of branch-and-cut algorithms. The ABACUS library is only used by branch-and-cut algorithms, whereas the whole basic functionality of AGD is independent of ABACUS. Therefore, we split the library into two parts, the basic part `AGD` and the part `AGDopt` that contains all ABACUS dependent classes. This makes it possible to use a subset of the algorithms in AGD without having an ABACUS installation – a LEDA installation is sufficient in this case.

Graph drawing algorithms that are tightly connected with a particular visualization component (e.g., a graph editor) or use very specialized data structures for representing a drawing (e.g., with many graphical attributes like line styles, text fonts, ...) are of limited use because it is difficult to integrate them into an application program. Each application is forced to support at least the same set of graphical attributes. Therefore, we decided to define a basic set of attributes which are required by graph drawing algorithms. An application must support these basic attributes, but can also use many more. Basic attributes of a node are the width and height of a rectangular box surrounding its graphical representation and the position of the center of this representation. Considering only the rectangular outline is convenient and sufficient for graph drawing algorithms. Basic attributes of an edge are simply the bend points of its line representation and the two anchor points connecting the line to its source and target nodes. The graph drawing algorithms in AGD access the basic attributes using a generic layout interface class. For a particular visualization component, an implementation is derived from the generic class and some virtual functions are overridden. The implementation class is responsible for storing the attributes. When a graph drawing algorithm is called, an object of this implementation class is passed and used by the algorithm to produce the layout.

An implementation of the generic layout interface for LEDA's graph editor `GraphWin` is already part of AGD, as well as a simple data structure for storing a layout. The latter is particularly useful for testing algorithms when it is not necessary to display the computed layout. AGD comes with two demo programs, `agd_demo` and `agd_opt_demo` based on `GraphWin`, that realize a graph editor with sophisticated layout facilities. Both programs allow to experiment with the various algorithms of AGD, i.e., changing options and using different algorithms for subproblems. They can also be extended and adapted by developers, since their source code is part of AGD. If an application program is not written in C++, it is not possible to use AGD algorithms via the library directly. A solution for such applications is provided by the program `agd_server` by Stefan Näher. The server allows to use AGD algorithms via a file or socket interface. The program reads the graph in GML format [Him97] from a given input file and loads the AGD options that specify the layout algorithm from a second file. Then, the selected algorithm is applied and the result is written back to the input file, again in GML format.

The following code fragment gives a programming example with AGD. It shows how to set the planarizer option for the planarization layout.

```
OptPlanarSubgraph optSub;
OneEdgeMinCrossInserter optInsert;
optInsert.removeReinsert(EdgeInsertionModule::all);

SubgraphPlanarizer planarizer;
planarizer.set_subgraph(optSub);
planarizer.set_inserter(optInsert);

PlanarizationLayout plan;
plan.set_planarizer(planarizer);
```

We use a subgraph planarizer called `planarizer` and set its subgraph option to a module for computing an optimal planar subgraph and its edge insertion option to the `OneEdgeMinCrossInserter` with the `removeReinsert` option set to `all`. Finally, we call the planarization layout algorithm `plan` for a graph `myGraph` with the layout information `myLayout` (containing information on the size of the vertices, the position of the vertices, and the position of the bend points):

```
plan.call(myGraph,myLayout).
```

Acknowledgements. The AGD system, as it is today, is by far not the product of the authors of this paper. It benefits from software contributions and advice of many additional supporters, in alphabetical order: David Alberts, Dirk Ambras, Ralf Brockenauer, Christoph Buchheim, Matthias Elf, Sergej Fialko, Karsten Klein, Gunter Koch, Michael Krüger, Thomas Lange, Dirk Lütke–Hüttmann, Stefan Näher, René Weiskircher, and Thomas Ziegler. We gratefully acknowledge the fruitful cooperation.

References

[AGD00] *AGD User Manual (Version 1.2)*, 2000. Max-Planck-Institut Saarbrücken, Technische Universität Wien, Universität zu Köln, Universität Trier. See also `http://www.mpi-sb.mpg.de/AGD/`.

[BDD$^+$00] S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal area drawings for orthogonal representations. *Computational Geometry Theory and Applications (CGTA)*, 2000. To appear.

[BJL00] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *LNCS*, pages 229–240. Springer-Verlag, 2000.

[BNT86] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data-flow diagrams. *IEEE Trans. Soft. Eng.*, SE-12(4):538–546, 1986.

[BTT84] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *J. Syst. and Softw.*, 4:163–173, 1984.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[CK97] M. Chrobak and G. Kant. Convex grid drawings of 3-connected planar graphs. *Internat. Journal on Computational Geometry and Applications*, 7(3):211–224, 1997.

[DPP90] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[DT96] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996.

[DTT92] G. Di Battista, R. Tamassia, and I. G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.*, 7:381–401, 1992.

[EK86] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21(A):89–98, 1986.

[EL95] P. Eades and X. Lin. A new heuristic for the feedback arc set problem. *Australian Journal of Combinatorics*, 12:15–26, 1995.

[EM99] P. Eades and P. Mutzel. Graph drawing algorithms. In M. Atallah, editor, *CRC Handbook of Algorithms and Theory of Computation*, chapter 9, pages 9–1–9–26. CRC Press, 1999.

[EW86] P. Eades and N. Wormald. The median heuristic for drawing 2-layers networks. Technical Report 69, Dept. of Comp. Sci., University of Queensland, 1986.

[Fia97] S. Fialko. Das planare Augmentierungsproblem. Master's thesis, Universität des Saarlandes, Saarbrücken, 1997.

[FK96] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F.J. Brandenburg, editor, *Graph Drawing '95)*, volume 1027 of *LNCS*, pages 254–266. Springer, 1996.

[FM98] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 260–269, San Francisco, California, 1998. ACM Press.

[FR91] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.

[GDT] Graph Drawing Toolkit: An object-oriented library for handling and drawing graphs. `http://www.dia.uniroma3.it/~gdt`.

[GJR85] M. Grötschel, M. Jünger, and G. Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33:28–42, 1985.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[GM97] C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.

[GM98] C. Gutwenger and P. Mutzel. Planar polyline drawings with good angular resolution. In S. Whitesides, editor, *Graph Drawing '98*, volume 1547 of *LNCS*, pages 167–182. Springer-Verlag, 1998.

[GM00] C. Gutwenger and P. Mutzel. A linear-time implementation of SPQR-trees. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *LNCS*, pages 77–90. Springer-Verlag, 2000.

[GMW01] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '2001)*, pages 246–255, Washington, DC, 2001. ACM Press.

[GT98] N. Gelfand and R. Tamassia. Algorithmic patterns for orthogonal graph drawing. In S. Whitesides, editor, *Graph Drawing '98*, volume 1547 of *LNCS*, pages 138–152. Springer-Verlag, 1998.

[Him97] M. Himsolt. GML: A portable graph file format. Technical report, Universität Passau, 1997. See also `http://www.uni-passau.de/Graphlet/GML`.

[JLM98] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Trans. on Computer-Aided Design*, 17(7), 1998.

[JM94] M. Jünger and P. Mutzel. The polyhedral approach to the maximum planar subgraph problem: New chances for related problems. In *DIMACS Graph Drawing '94*, volume 894 of *LNCS*, pages 119–130. Springer-Verlag, 1994.

[JM96a] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms and Applications (JGAA)* (`http://www.cs.brown.edu/publications/jgaa/`), 1(1):1–25, 1996.

[JM96b] M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

[JT00] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut and price algorithms in integer programming and combinatorial optimization. *Software – Practice and Experience*, 30:1325–1352, 2000.

[Kan96] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica, Special Issue on Graph Drawing*, 16(1):4–32, 1996.

[KKM01] G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Graph Drawing (Proc. 2000)*, LNCS. Springer Verlag, 2001.

[KM98] G. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max–Planck–Institut f. Informatik, Saarbrücken, 1998.

[KM99] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In G. P. Cornuéjols, R. E. Burkard, and G. J. Woeginger, editors, *Integer Programming and Combinatorial Optimization (IPCO '99)*, volume 1610 of *LNCS*, pages 304–319. Springer, 1999.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, New York, 1990.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[MSM99] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer crossing minimization. In J. Kratochvil, editor, *Graph Drawing '99*, volume 1731 of *LNCS*, pages 217–224. Springer-Verlag, 1999.

[Mut95] P. Mutzel. A polyhedral approach to planar augmentation and related problems. In Paul Spirakis, editor, *Annual European Symposium on Algorithms (ESA-3) : Corfu, Greece, September 2 5-27, 1995; proceedings*, volume 979 of *LNCS*, pages 494–507, Berlin, 1995. Springer.

[RT81] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.

[RT86] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.*, 1(4):343–353, 1986.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[Tam87] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[TBB88] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(3):743–768, 1963.

[Wal90] J. Q. Walker II. A node-positioning algorithm for general trees. *Software – Practice and Experiments*, 20(7):685–705, 1990.

An Overview of the GXL Graph Exchange Language[⋆]

Andreas Winter, Bernt Kullbach, and Volker Riediger

Universität Koblenz-Landau
Institut für Softwaretechnik
D-56016 Koblenz, Postfach 201602
`mailto:(winter|kullbach|riediger)@uni-koblenz.de`
`http://www.gupro.de/(winter|kullbach|riediger)`

Abstract.

GXL (Graph eXchange Language) is designed to be a standard exchange format for graph-based tools. GXL is defined as an XML sublanguage, which offers support for exchanging instance graphs together with their appropriate schema information in a uniform format. Formally, GXL is based on typed, attributed, ordered directed graphs, which are extended by concepts to support representing hypergraphs and hierarchical graphs. Using this general graph model, GXL offers a versatile support for exchanging nearly all kinds of graphs.
This report intends to give a short overview on the main features of GXL.

1 Motivation and Background

A great variety of software tools relies on graphs as internal data representation. A standardized language for exchanging those graphs offers a first step in improving *interoperability* between these tools. For instance, a common graph interchange format allows building a powerful reverse engineering workbench. Such a reverse engineering workbench composes various graph-based tools like extractors (e. g. scanner, parser), abstractors (e. g. query tools, structure recognition tools, slicing tools etc.), and visualizers (e. g. graph and diagram visualizer, code browser). [22] gives an overview on existing combinations of tool components used in various reverse engineering projects.

The development of *GXL (Graph eXchange Language)* aims at supporting data interoperability between reverse engineering tools. GXL was ratified as *standard exchange format* in reverse engineering at the Dagstuhl Seminar "Interoperability of Reverse Engineering Tools" in January 2001 [4]. But since GXL was developed as a general format for describing graph structures, it is applicable in further areas of tool interoperability. Especially, GXL is used to define the graph part in the exchange format GTXL (Graph Transformation eXchange Language) [17], [34].

[⋆] This paper is an extended abstract of [37].

GXL originated in a merger of *GRAph eXchange format (GraX)* [7], *Tuple Attribute Language (TA)* [21], and the graph format of the *PROGRES* graph rewriting system [32]. The graph models used here were supplemented by additional concepts to handle hierarchical graphs and hypergraphs. Furthermore, GXL includes ideas from common exchange formats used in reverse engineering, including *Relation Partition Algebra (RPA)* [27] and *Rigi Standard Format (RSF)* [38]. The development of GXL was also influenced by various formats used in graph drawing, e. g. *daVinci* [10], *GML* [16], *XGMML (eXtensible Graph Markup and Modeling Language)* [39], and *GraphXML* [20]. Thus, GXL covers most of the important graph formats. GXL can be viewed as a generalization of these formats.

Exchanging graphs with GXL deals with both *instance graphs* and their corresponding *graph schemas.* Firstly, GXL offers a versatile support for exchanging all kinds of graphs based on *typed, attributed, directed, ordered graphs* including *hypergraphs* and *hierarchical graphs.* Secondly, GXL offers means for exchanging graph schemas representing the graph structure, i. e. the definition of node and edge types, their attribute schemas and their incidence structure. Both, instance graphs and graph schemas, are exchanged by XML documents (Extended Markup Language) [35].

This paper introduces into the basic concepts of GXL version 1.0 for exchanging instance graphs (cf. section 2) and graph schemas (cf. section 3). The language definition of GXL is given by its XML document type definition (DTD). Section 4 summarizes the current usage of GXL.

A more comprehensive description of GXL is given in [37]. Up-to-date information including tutorials and further GXL documents are collected at `http://www.gupro.de/GXL`.

2 Exchanging Graphs

Due to their mathematical foundation and algorithmic power, graphs are a common data structure in software engineering. Different graph models, e. g. directed graphs, undirected graphs, node attributed graphs, edge attributed graphs, node typed graphs, edge typed graphs, ordered graphs, relational graphs, acyclic graphs, trees, etc. or combinations of these graph models are utilized in many software systems. To support interoperability of graph based tools, the underlying graph model has to be as rich as possible to cover most of these graph models.

Such a common graph model is given by *typed, attributed, directed, ordered graphs (TGraphs)* [6], [7]. TGraphs are *directed* graphs, whose nodes and edges may be *attributed* and *typed.* Each type can be assigned an individual attribute schema specifying the possible attributes of nodes and edges. Furthermore, TGraphs are *ordered*, i. e. the node set, the edge set, and the sets of edges incident to a node have a total ordering. This ordering gives modeling power to describe sequences of objects (e. g. parameter lists) and facilitates the implementation of deterministic graph algorithms. In applying TGraphs to the sketched graph models, not all properties of TGraphs have to be used to their full extent. These graph models can be viewed as specializations of TGraphs. Exchanging TGraphs with GXL is introduced in section 2.1

To offer support for *hypergraphs* and *hierarchical graphs*, TGraphs were extended by n-ary edges and by nodes and edges containing lower level graphs. GXL language constructs for exchanging those extended graphs are sketched in section 2.2. The complete GXL language definition in terms of an XML document type definition is given in section 2.3.

2.1 Exchanging Typed, Attributed, Directed, Ordered Graphs

The UML object diagram (cf. [31]) in figure 1 shows a node and edge typed, node and edge attributed, directed, ordered graph representing a program fragment on ASG (abstract syntax graph) level. Function *main* calls function $a = max(a, b)$ in line *8* and function $b = min(b, a)$ in line *19*.

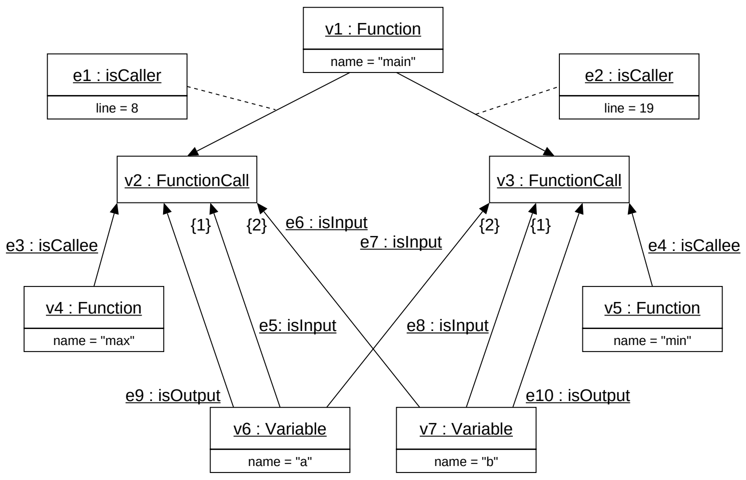


Fig. 1. Typed, attributed, directed, ordered graph

The functions *main*, *max* and *min* are represented by nodes of type *Function*. These nodes are attributed with the functions' name. *FunctionCall* nodes represent the calls of functions *max* and *min*. *FunctionCall* nodes are associated to the caller by *isCaller* edges and to the callee by *isCallee* edges. *isCaller* edges are attributed with a line attribute showing the line number which contains the call. Input parameters (represented by *Variable* nodes that are attributed with the variables' name) are associated by *isInput* edges. The ordering of parameter lists is given by ordering the incidences of *isInput* edges pointing to *FunctionCall* nodes. The first edge of type *isInput* incident to function call *v2* (modeling the call of *max(a,b)*) comes from node *v6* representing variable *a*. The second edge of type *isInput* connects to the second parameter *b* (node *v7*). The incidences of *isInput* edges associated with node *v3* model the reversed parameter order. Output parameters are associated to their function calls by *isOutput* edges.

Exchanging graphs like the one in figure 1 requires language constructs for representing nodes, edges and their incidence relation. Furthermore, support for describing type information and attribute values is needed.

```
<?xml version = "1.0" ?>
<!DOCTYPE gxl
      SYSTEM "gxl-1.0.dtd">
<gxl>
<graph id = "simpleGraph"
               edgeids = "true">
   <type xlink:href =
   "schema.gxl#Schema"/>
   <node id = "v1" >
      <type xlink:href =
      "schema.gxl#Function"/>
      <attr name = "name" >
         <string>main</string>
      </attr>
   </node>
   <node id = "v2" >
      <type xlink:href =
      "schema.gxl#FunctionCall"/>
   </node>
   <node id = "v3" >
      <type xlink:href =
      "schema.gxl#FunctionCall"/>
   </node>
   <node id = "v4" >
      <type xlink:href =
      "schema.gxl#Function"/>
      <attr name = "name" >
         <string>max</string>
      </attr>
   </node>
   <node id = "v5" >
      <type xlink:href =
      "schema.gxl#Function"/>
      <attr name = "name" >
         <string>min</string>
      </attr>
   </node>
<node id = "v6" >
   <type xlink:href =
   "schema.gxl#Variable"/>
   <attr name = "name" >
      <string>a</string>
   </attr>
</node>
<node id = "v7" >
   <type xlink:href =
   "schema.gxl#Variable"/>
   <attr name = "name" >
      <string>b</string>
   </attr>
</node>
<edge id = "e1"
   from = "v2" to = v1">
   <type xlink:href =
   "schema.gxl#isCaller"/>
   <attr name = "line" >
      <int>8</int>
   </attr>
</edge>
<edge id = "e2"
   from = "v3" to = v1">
   <type xlink:href =
   "schema.gxl#isCaller"/>
   <attr name = "line" >
      <int>19</int>
   </attr>
</edge>
<edge id = "e3"
   from = "v4" to = v2">
   <type xlink:href =
   "schema.gxl#isCallee"/>
</edge>
<edge id = "e9"
   from = "v6" to = v2"
   <type xlink:href =
   "schema.gxl#isOutput">
</edge>
<edge id = "e5"
   from = "v6" to = v2"
   toorder = "1">
   <type xlink:href =
   "schema.gxl#isInput"/>
</edge>
<edge id = "e6"
   from = "v7" to = v2"
   toorder = "2">
   <type xlink:href =
   "schema.gxl#isInput"/>
</edge>
<edge id = "e7"
   from = "v6" to = v3"
   toorder = "2">
   <type xlink:href =
   "schema .gxl#isInput"/>
</edge>
<edge id = "e8"
   from = "v7" to = v3"
   toorder = "1">
   <type xlink:href =
   "schema.gxl#isInput"/>
</edge>
<edge id = "e9"
   from = "v6" to = v2"
   <type xlink:href =
   "schema.gxl#isOutput">
</edge>
<edge id = "e10"
   from = "v7" to = v3"
   <type xlink:href =
   "schema.gxl#isOutput">
</edge>
</graph>
</gxl>
```

Fig. 2. GXL representation of graph from figure 1

Figure 2 depicts the graph from figure 1 as GXL document. XML documents start with specifying the XML version and the underlying document type definition, here "gxl-1.0.dtd" (cf. figure 3). The body of a GXL document is enclosed in <**gxl**> tags. The GXL document in figure 2 contains a graph with a unique identifier "simpleGraph". The graph refers to its associated graph schema object Schema (cf. section 3) stored in file schema.gxl.

Nodes and edges of a given graph are depicted by <**node**> and <**edge**> elements which can be addressed by their id attribute. Incidence information of edges including edge orientation is stored in from and to attributes within <**edge**> tags. Ordering of incidences is also modeled here. Attributes fromorder and toorder represent the position of an edge in the incidence list of its start and target node.

Node and edge types are represented by links pointing to the appropriate schema information. These links are enclosed in <**type**> elements.

<**node**> and <**edge**> elements may additionally contain further attribute information. <**attr**> elements describe attribute names and values. Like OCL [36], GXL provides <**bool**>, <**int**>, <**float**>, and <**string**> attributes. Furthermore, enumeration values (<**enum**>) and URI-references (<**locator**>) to externally stored objects are supported. Attribute values might be substructured. Here, GXL offers composite attributes like sequences (<**seq**>), sets (<**set**>), multi sets (<**bag**>), and tuples (<**tup**>).

2.2 Exchanging Extended Graphs

In addition to typed, attributed, ordered, directed graphs, GXL provides the exchange of *hypergraphs* and *hierarchical graphs.*

Hypergraphs contain n-ary edges (hyperedges) connecting not only two adjacent nodes. *Hyperedges* are exchanged by <**rel**> elements, containing references to the incident graph objects. These references (tentacles) are stored in <**relend**> elements (relation end).

Edges can be viewed as 2-ary hyperedges. Thus, in GXL, edge information can be represented by *binary hyperedges.* Since graphs with (binary) edges are widespread in software engineering and most applications deal with graphs instead of hypergraphs, GXL offers both, the element <**edge**> for exchanging (binary) edges and and the element <**rel**> for hyperedges.

Like binary edges, tentacles may be directed or undirected as well as ordered. The ordering of tentacles incident to their target object and the ordering of tentacles with respect to their hyperedge object is represented analogously to the ordering of incident edges by using XML attributes.

Hierarchical graphs are graphs where nodes, edges, and hyperedges contain further graphs. GXL supports exchanging hierarchical graphs by nesting those inner graphs as <**graph**> elements in their enclosing node, edge, and hyperedge representation.

2.3 GXL Document Type Definition

The language features of GXL for exchanging typed, attributed, directed, ordered graphs (cf. section 2.1) and extended graphs (cf. section 2.2) are summarized in a conceptual model defining the graph model supported by GXL. The GXL graph model is completely described at `http://www.gupro.de/GXL/` (graph model) with its graph structure part and its attribute part.

Since GXL is an XML sublanguage, the GXL graph model had to be transcribed into an XML document type definition (DTD) or an appropriate XML schema definition. To keep GXL simple and less verbose, this translation was done by hand. The resulting DTD (cf. figure 3, a commented version is given at `http://www.gupro.de/GXL` (DTD)) requires only 18 XML elements. In contrast, an appropriate DTD generated with IBMs XMI Toolkit [23] according the XML Metadata Interchange (XMI) principles for developing DTDs [26, section 3] requires 66 elements for the GXL core and and additional 63 elements for XMI and Corba related aspects.

```
<!– extensions –>
<!ENTITY % gxl-extension         "" >
<!ENTITY % graph-extension       "" >
<!ENTITY % node-extension        "" >
<!ENTITY % edge-extension        "" >
<!ENTITY % rel-extension         "" >
<!ENTITY % value-extension       "" >
<!ENTITY % relend-extension      "" >
<!ENTITY % gxl-attr-extension    "" >
<!ENTITY % graph-attr-extension  "" >
<!ENTITY % node-attr-extension   "" >
<!ENTITY % edge-attr-extension   "" >
<!ENTITY % rel-attr-extension    "" >
<!ENTITY % relend-attr-extension "" >

<!– attribute values –>
<!ENTITY % val " locator | bool | int |
                 float | string | enum |
                 seq | set | bag | tup
                 % value-extension;" >

<!– gxl –>
<!ELEMENT gxl (graph* %gxl-extension;) >
<!ATTLIST gxl
    xmlns:xlink CDATA        #FIXED
                "www.w3.org/1999/xlink"
    %gxl-attr-extension; >

<!– type –>
<!ELEMENT type EMPTY>
<!ATTLIST type
    xlink:type  (simple)     #FIXED "simple"
    xlink:href  CDATA        #REQUIRED >

<!– graph –>
<!ELEMENT graph (type? , attr* ,
                ( node | edge | rel )*
                %graph-extension;) >
<!ATTLIST graph
    id          ID           #REQUIRED
    role        NMTOKEN      #IMPLIED
    edgeids     ( true | false ) "false"
    hypergraph  ( true | false ) "false"
    edgemode    ( directed | undirected |
                defaultdirected |
                defaultundirected)
                "directed"
    %graph-attr-extension; >

<!– node –>
<!ELEMENT node (type? , attr*, graph*
                %node-extension;) >
<!ATTLIST node
    id          ID           #REQUIRED
    %node-attr-extension; >

<!– edge –>
<!ELEMENT edge (type?, attr*, graph*
                %edge-extension;) >
<!ATTLIST edge
    id          ID                 #IMPLIED
    from        IDREF              #REQUIRED
    to          IDREF              #REQUIRED
    fromorder   CDATA              #IMPLIED
    toorder     CDATA              #IMPLIED
    isdirected  ( true | false )   #IMPLIED
    %edge-attr-extension; >

<!– rel –>
<!ELEMENT rel (type? , attr*, graph*, relend*
                %rel-extension;) >
<!ATTLIST rel
    id          ID                 #IMPLIED
    isdirected  ( true | false )   #IMPLIED
    %rel-attr-extension; >

<!– relend –>
<!ELEMENT relend (attr* %relend-extension;) >
<!ATTLIST relend
    target      IDREF              #REQUIRED
    role        NMTOKEN            #IMPLIED
    direction   ( in | out | none) #IMPLIED
    startorder  CDATA              #IMPLIED
    endorder    CDATA              #IMPLIED
    %relend-attr-extension; >

<!– attr –>
<!ELEMENT attr (type?, attr*, (%val;)) >
<!ATTLIST attr
    id          IDREF              #IMPLIED
    name        NMTOKEN            #REQUIRED
    kind        NMTOKEN            #IMPLIED >

<!– locator –>
<!ELEMENT locator EMPTY >
<!ATTLIST locator
    xlink:type  (simple)           #FIXED "simple"
    xlink:href  CDATA              #IMPLIED >

<!– attribute values –>
<!ELEMENT bool (#PCDATA) >
<!ELEMENT int (#PCDATA) >
<!ELEMENT float (#PCDATA) >
<!ELEMENT string (#PCDATA) >
<!ELEMENT enum (#PCDATA) >
<!ELEMENT seq (%val;)* >
<!ELEMENT set (%val;)* >
<!ELEMENT bag (%val;)* >
<!ELEMENT tup (%val;)* >
```

Fig. 3. GXL Document Type Definition

3 Exchanging Graph Schemas

Graphs only offer a plain structured means for describing objects (nodes) and their interrelationship (edges, hyperedges). Graphs have no meaning of their own. The meaning of graphs corresponds to the context in which they are used and exchanged. The application and interchange context determines

– which node, edge, and hyperedge types are used,
– how nodes, edges, and hyperedges of given types are related,
– which attribute structures are associated to nodes, edges, and hyperedges, and
– which additional constraints (like ordering of incidences, degree-restrictions etc.) have to be complied.

This schematic data can be described by *conceptual modeling techniques*. Class diagrams offer a suited declarative language to define graph classes with respect to a given application or interchange context [7].

3.1 Describing Graph Classes by UML Class Diagrams

In GXL, graph classes are defined by UML class diagrams [31]. Figure 4 shows a graph schema defining classes of graphs like the one given in figure 1. Node classes (*FunctionCall*, *Function*, and *Variable*) are defined by classes. Edge classes (*isCallee*, *isInput*, and *isOutput*) are defined by associations. Attributed edge classes (*isCaller*) are described by association classes. Like classes, they contain the associated attribute structures. The orientation of edges is depicted by a filled triangle (cf. [31, p. 155]. Multiplicities denote degree restrictions. Ordering of incidences is indicated by the keyword {ordered}.

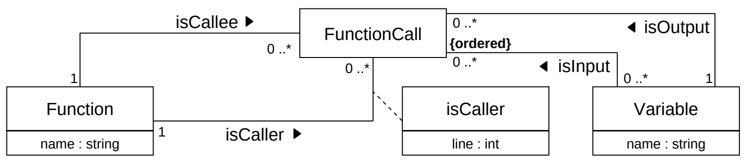


Fig. 4. Graph schema (UML class diagram)

In a similar way, UML class diagrams offer language constructs to model classes of hyperedges (diamonds) and classes of attributed hyperedges (diamonds with an associated class). The definition of hierarchical graphs requires an additional language construct representing graph classes themselves. This is done by <<GraphClass>> stereotypes.

To offer up-to-date conceptual modeling power, the GXL schema notation provides generalization of node-, edge-, and hyperedge classes as well as aggregation and composition by using the appropriate UML notation.

3.2 Describing Graph Classes by Graphs

Since UML class diagrams are structured information themselves, they may be represented as graphs as well. For exchanging graph schemas in GXL, UML class diagrams are transfered into equivalent graph representations. Thus, instance graphs and schemas are exchanged with the *same type of document*, i. e. XML documents matching the GXL DTD (cf. section 2.3).

In contrast to the strategy proposed by XML Meta Data Interchange (XMI) [26], GXL schemas are *not* exchanged by XML documents according to the Meta Object Facility (MOF) [25]. XMI/MOF offers a general, but very verbose format for exchanging UML class diagrams as XML streams. By generating individual document type definitions to a given UML class diagram, it also supports exchanging instance graphs as XML documents. Next to its exaggerated verbosity, which contradicts the requirement for exchange formats of as compact as possible documents, the XMI/MOF approach requires *different types of documents* for representing schema and instance graphs. Especially in applications dealing with schema information on instance level (e. g. in tools for editing and analyzing schemas), this leads to the disadvantage of different documents representing the same information, one on instance level (as XML document) and one on schema level (as XML DTD). The GXL approach treats schema and instance information in exactly the same way. Schema and instance graphs are exchanged according to the DTD given in Figure 3.

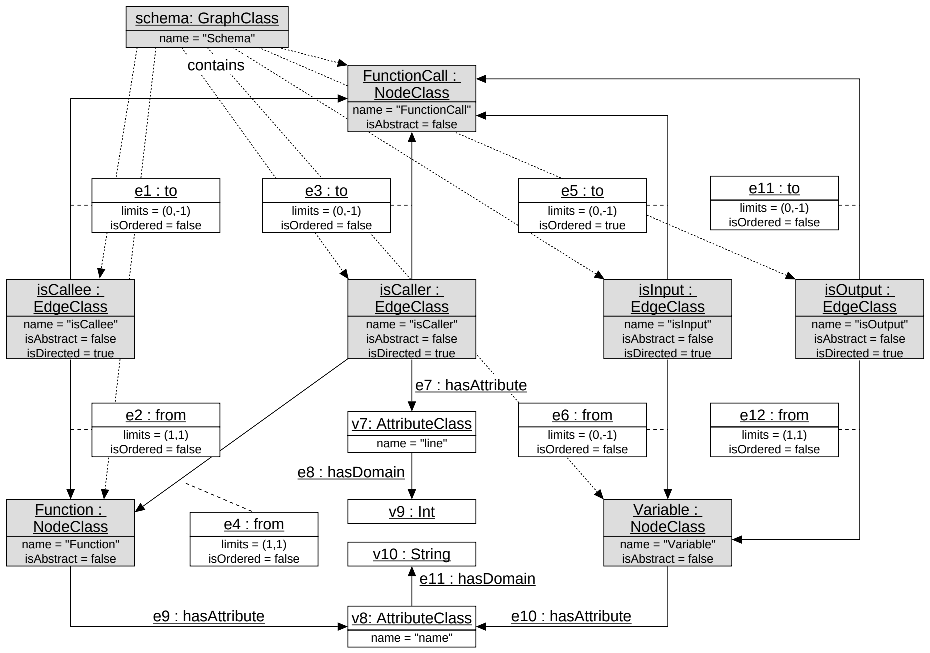


Fig. 5. Graph schema (schema graph)

Figure 5 depicts the transformation of the class diagram in figure 4 into a node and edge typed, node and edge attributed, directed graph. Node classes, edge classes, attributes and their domains are modeled by nodes of suitable node types. Their attributes describe further properties. Interrelationships between surrogates of these classes are represented by edges of proper types. Attribute information is associated with surrogates of node classes, edge classes and associations by *hasAttribute* and *hasDomain* edges. *from* and *to* edges model incidences of associations including their orientation. Multiplicities of associations are stored in limits-attributes. The boolean attribute isOrdered indicates ordered incidences.

Further attribute types and extended concepts like graph hierarchy, classes of hyperedges, aggregation and composition, generalization and default attribute values are modeled analogously.

GXL documents, representing instance graphs of a given graph schema refer to those nodes of the equivalent schema graph representing node classes (*NodeClass*) and edge classes (*EdgeClass*). The graph class itself is represented by a *GraphClass* node. This node is connected by *contains* edges to all surrogates of node and edge classes defined in this graph class. Schema references in GXL-documents refer to these *GraphClass* nodes in GXL schema graphs (cf. the type element of graph simpleGraph in figure 2). In figure 5 nodes representing these class definitions are shaded. These items are refered to by the instance graph in figure 1.

GXL views edges as *first class objects* which have their own identity, might be typed and attributed, and might be included in a generalization hierarchy. Thus, surrogates of associations and associated classes have to be connected to further information. For generality and simplicity reasons GXL schema graphs are restricted to ordinary typed, attributed, directed graphs. Hence, this edge-like information is represented by nodes as well. Although the GXL DTD provides edges connecting edges, GXL schema graphs do not use this feature.

The graph class of correct GXL schema graphs is represented as a GXL schema. A UML diagram representing this *GXL metaschema* is presented with its graph part, its attribute part, and its value part at `http://www.gupro.de/GXL/` (meta schema).

Each UML class diagram defining a GXL graph schema can be represented by a graph (schema graph) matching the GXL metaschema. Thus, schema graphs are instances of the GXL metaschema. They are exchanged like all instance graphs (cf. section 2) referring to a GXL document, here representing the GXL metaschema. Since the schema graph representing the GXL metaschema is an instance of itself, it is exchanged by a self referring GXL document.

4 Using GXL

At the Dagstuhl seminar on "Interoperability of Reverse Engineering Tools" GXL version 1.0 was ratified as the *standard exchange format* in reverse engineering [4]. Currently, various groups in software (re)engineering are implementing

GXL import and export facilities to their tools (e. g. Bauhaus [1], Columbus [8], CPPX [3], Fujaba [11], GUPRO [18], PBS [28], RPA (Philips Research), PROGRES [29], Rigi [30], Shrimp [33]). Others are going to implement tools to support working with GXL. For instance, a framework for GXL Converters [12] and an XMI2GXL translator [40] were developed at Univ. BW München. Further activities deal with providing graph query machines (GReQL, Univ. Koblenz) to GXL graphs or GXL-based graph databases (Univ. Aachen).

An important feature of GXL is its support for exchanging schema information. Based on this capability, reference schemas for certain standard applications in reverse engineering are currently under development. These activities address reference schemas for data reverse engineering (DRE, Univ. Namur, Paderborn, Victoria), the Dagstuhl Middle Model [24] or abstract syntax graph models for C++ [3], [9].

Furthermore, groups developing graph transformation tools (e. g. GenSet [15], PROGRES [29]) or graph visualization tools (e. g. GVF [19], Shrimp [33], yFiles [41]) already use GXL or pronounced to use GXL. At University of Toronto, GXL is applied within an undergraduate software engineering course to create a graph editor/layouter [5].

GXL also serves as foundation to define further graph oriented exchange formats. Thus, GXL defines the graph part in the exchange format GTXL (Graph Transformation eXchange Language) [17], [34]. Activities in the graph drawing community also deal with the development of an exchange format for graph layout [13]. In a panel on graph exchange formats at Graph Drawing 2001 in Vienna [14] GXL and GraphML [2] were discussed and compared. There is evidence of combining the structure part of GXL with the graph layout part and the modularization part of GraphML to form a general and comprehensive graph exchange format.

5 Conclusion

The previous sections gave a short introduction in the GXL Graph eXchange Language version 1.0 and its current applications.

Summarizing, GXL offers an already widely used XML sublanguage for interchanging typed, attributed, directed ordered graphs including hypergraphs and hierarchical graphs including their appropriate schemas. By focusing on graph structure, GXL provides the core for defining a family of special suited graph exchange formats.

Acknowledgment. We would like to thank the GXL co-authors Richard C. Holt, Andy Schürr, and Susan Sim for various fruitful discussions on the development of GXL, and Oliver Heinen and Kevin Hirschmann for realizing the GXL web-site and for implementing the GUPRO related GXL tools. Thanks to Rainer Koschke for many interesting discussions on interchange formats in reverse engineering, and some helpful remarks to improve this paper. Thanks to

all the users of GXL, who currently applying and testing GXL 1.0 in their tools. Their experience will be a significant aid to improve GXL.

References

1. Bauhaus: Software Architecture, Software Reengineering, and Program Understanding. `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/` (01.09.2001).
2. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marschall. GraphML Progress Report, Structural Layer Proposal. In *to appear: Graph Drawing 2001 (Proceedings)*. 2001.
3. CPPX: Open Source C++ Fact Extractor. `http://swag.uwaterloo.ca/~cppx/` (01.09.2001).
4. J. Ebert, K. Kontogiannis, J. Mylopoulos: Interoperability of Reverse Engineering Tools. `http://www.dagstuhl.de/DATA/Reports/01041/` (18.04.2001), 2001.
5. S. Easterbrook. CSC444F: Software Engineering I (Fall Term 2001), University of Toronto. `http://www.cs.toronto.edu/~sme/CSC444F/` (15.09.2001), 2001.
6. J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In *E. Mayr, G. Schmidt, and G. Tinhofer, editors.* Graphtheoretic Concepts in Computer Science*, LNCS 903. Springer, Berlin*, pages 38–50. 1995.
7. J. Ebert, B. Kullbach, and A. Winter. GraX – An Interchange Format for Reengineering Tools. In *Sixth Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, pages 89–98, 1999.
8. R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings SPLST 2001*, Szeged, Hungary (`http://www.inf.u-szeged.hu/~ferenc/research/ferencr_columbus.pdf`, (01.09.2001)), pages 16–27. June 2001.
9. R. Ferenc, S. Elliott Sim, R. C. Holt, R. Koschke, and T. Gyimòthy. Towards a Standard Schema for C/C++. In *Eighth Working Conference on Reverse Engineering.* IEEE Computer Society, Los Alamitos, pages 49–58. 2001.
10. M. Fröhlich and M. Werner. daVinci V2.0.x Online Documentation. `http://www.tzi.de/~davinci/docs/` (18.04.2001), June 1996.
11. Fujaba: From UML to Java and back again. `http://www.uni-paderborn.de/cs/fujaba/` (01.09.2001).
12. GCF - a GXL Converter Framework. `http://www2.informatik.unibw-muenchen.de/GXL/triebsees/index.htm` (01.09.2001).
13. Satellite Workshop on Data Exchange Formats 8th Int. Symposium on Graph Drawing (GD 2000). `http://www.cs.virginia.edu/~gd2000/gd-satellite.html` (14.09.2001), 2001.
14. Graph Drawing (GD 2001), Vienna. `http://www.ads.tuwien.ac.at/gd2001/` (06.12.2001), September 23.-26., 2001.
15. GenSet: Design Information Fusion. `http://www.cs.uoregon.edu/research/perpetual/dasada/Software/GenSet /index.html` (01.09.2001).
16. The GML File Format. `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/index.html` (18.04.2001).

17. Graph Transformation System Exchange Language. `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html` (18.08.2001).
18. GUPRO: Generic Understanding of Programs. `http://www.gupro.de/` (01.09.2001).
19. GVF - The Graph Visualization Framework . `http://www.cwi.nl/InfoVisu/` (01.09.2001).
20. I. Herman and M. S. Marshall. Graph XML – An XML based graph interchange format. Report INS-0009, Centrum voor Wiskunde en Informatica, Amsterdam, April 2000.
21. R. C. Holt. An Introduction to TA: The Tuple-Attribute Language. `http://plg.uwaterloo.ca/~holt/papers/ta.html` (18.4.2001), 1997.
22. R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Seventh Working Conference on Reverse Engineering*. IEEE Computer Society, Los Alamitos, pages 162–171. 2000.
23. XMI Toolkit 1.15 (Updated on: 25.04.2000). `http://alphaworks.ibm.com/tech/xmitoolkit` (01.09.2001), 2000.
24. T. Lethbridge, E. Plödereder, S. Tichelar, C. Riva, and P. Linos. The Dagstuhl Middle Level Model (DMM). internal note, 2001.
25. Meta Object Facility (MOF) Specification. `http://www.omg.org/technology/documents/formal/mof.htm` (02.09.2001), March 2000.
26. XML Meta Data Interchange (XMI) Specification. `http://www.omg.org/technology/documents/formal/xmi.htm` (01.09.2001), November 2000.
27. R. Ommering, L. van Feijs, and R. Krikhaar. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.
28. PBS: The Portable Bookshelf. `http://swag.uwaterloo.ca/pbs/` (01.09.2001).
29. A Graph Grammar Programming Environment - PROGRES. `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html` (01.09.2001).
30. RIGI: a visual tool for understanding legacy systems. `http://www.rigi.csc.uvic.ca/` (01.09.2001).
31. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, 1999.
32. A. Schürr, A. J. Winter, and A. Zündorf. PROGRES: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars: Applications, Languages, and Tools*, volume 2, World Scientific, Singapore, pages 487–550. 1999.
33. ShriMP Views: simple Hierarchical Multi-Perspective. `http://www.shrimpviews.com/` (01.09.2001).
34. G. Taenzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In *Proceedings UNIGRA satellite workshop of ETAPS'01*. 2001.
35. Extensible Markup Language (XML) 1.0. W3C recommendation, W3C XML Working Group, `http://www.w3.org/XML/` (17.04.2001), February 1998.
36. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language : Precise Modeling With UML*. Addison-Wesley, 1998.
37. A. Winter. Exchanging Graphs with GXL. In *to appear: Graph Drawing 2001 (Proceedings)*. 2001.
38. K. Wong. RIGI User's Manual, Version 5.4.4. `http://www.rigi.csc.uvic.ca/rigi/` (18.04.2001), 30. June 1998.

39. Extensible Graph Markup and Modeling Language). http://www.cs.rpi.edu/~puninj/XGMML/ (19.08.2001), 2001.
40. XIG - An XSLT-based XMI2GXL-Translator. http://ist.unibw-muenchen.de/GXL/volk/index.htm (01.09.2001).
41. yFiles - Interactive Visualization of Graph Strucutres. http://www-pr.informatik.uni-tuebingen.de/yfiles/ (01.09.2001).

Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications

Alexander A. Evstiougov-Babaev

AbsInt Angewandte Informatik GmbH,
Saarbrücken, Germany
http://www.absint.com
alex@absint.com

Abstract.

When working with complex software, visualization improves understanding considerably. Thus, enhancing the ability of programmers to picture the relationships between components in a complex program not only saves time but becomes progressively mission-critical with increasing software complexity.
aiCall is a software visualization tool which helps programmers to better understand their software, generally improving learning, speeding up development and saving considerable effort and expense. aiCall visualizes the call graph and the control flow graph of embedded application code. Currently supported targets are Infineon C16x [EBF01, Inf97] and STMicroelectronics ST10. These microcontroller families are very popular and widely used in consumer goods (cellular phones, CD-players, washing machines) and in safety-critical environments (airbags, navigation systems, and automotive controls).

1 Introduction

The complexity of embedded software increases continuously. Typical applications have to handle many sources of inputs which often requires complex interrupt handling code. Furthermore, embedded applications are usually time-critical and/or safety-critical. Due to the "embedded" aspect the use of debuggers is often restricted.

The source code of an application basically contains all the information about execution behavior. However, trying to understand the program at the textual level is usually a very tedious and time-consuming task, even though "intelligent" editors and CASE tools may be helpful.

Advanced static program analysis tools can help the developers of embedded systems to master the increasing complexity [Mar98]. There are natural program representations that can be automatically computed from the source code, e.g. the call graph and control flow graph. These graphs are often used in the intermediate program representation in compilers [WM95,ASU86]. Owing to the tremendous development of graph layout techniques in recent years and the availability of powerful graph browsers, it is now possible to graphically visualize the call graph and control flow graph for the program developer.

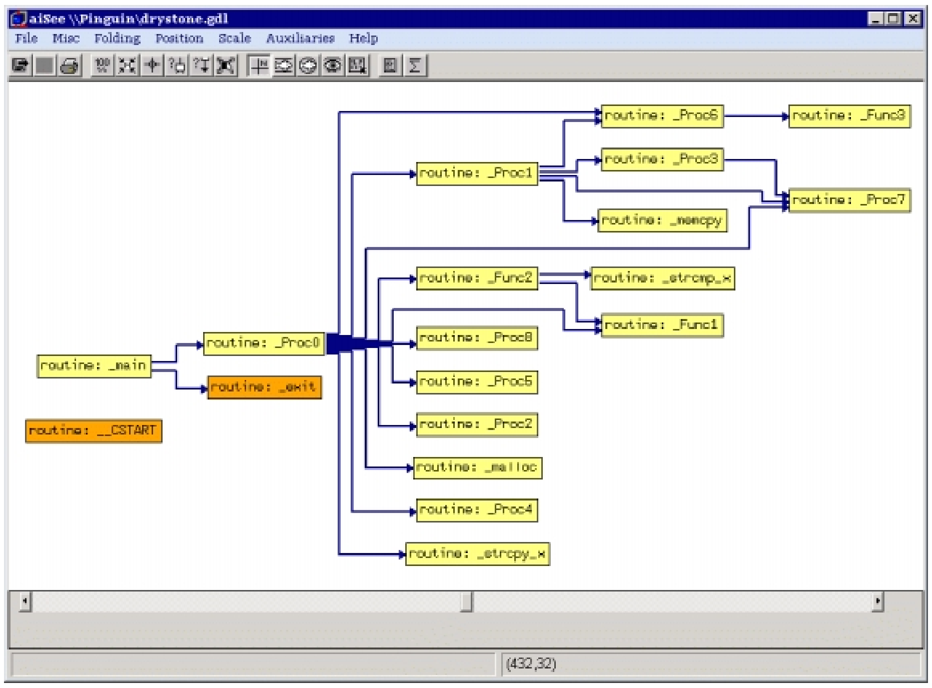


Fig. 1. Call graph of the Dhrystone benchmark application (Manhattan Layout, graph orientation left-to-right)

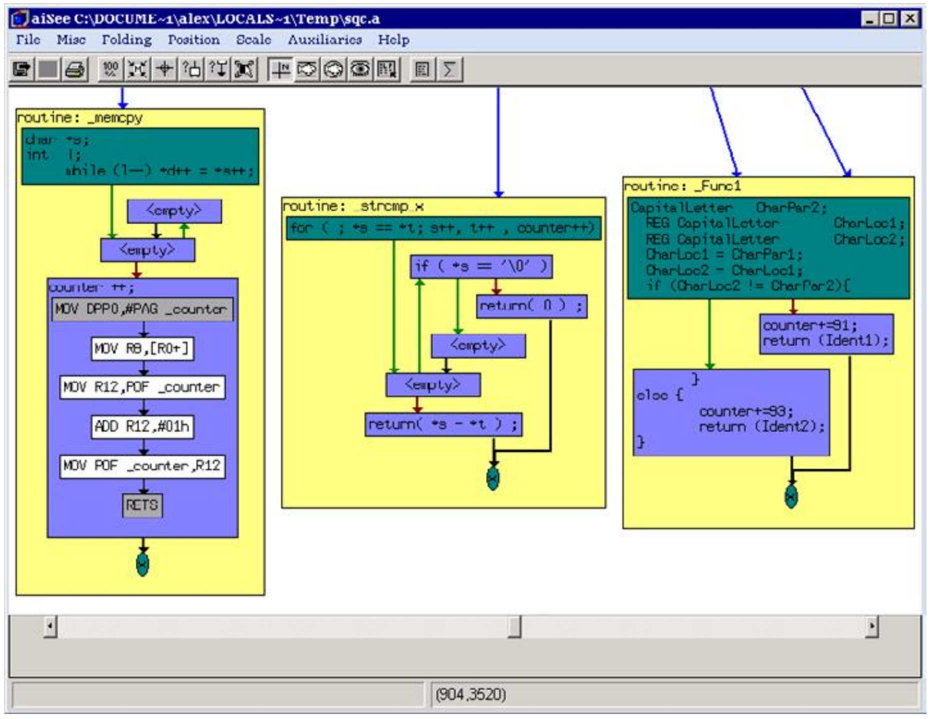
2 Brief Description of aiCall

aiCall analyzes the input assembly files in .src format as produced by the Tasking C compiler for C16x/ST10 and produces a textual and human-readable representation of the call graph and the control flow graph in GDL (graph description language [Abs,EB]). The integral parts of aiCall are:

– Frontend driver program: provides the graphical user interface and allows .src files to be selected for visualization.
– Translation program src2crl: reads the input assembly files and produces an intermediate representation of the call graph and control flow graph in the control flow representation language CRL[Lan99].

– Optional static program analysis modules, e.g. the Stack Usage Analysis Module, which automatically calculates the stack usage of application tasks.
– Translation program crl2gdl: reads the CRL file and produces a GDL specification of the call graph and the control flow graph.
– Backend: Graph layout software aiSee [aGVS].

aiSee calculates a customizable graph layout, which is then displayed and can be interactively explored and printed. aiSee implements fast layout calculation and convenient browsing of huge graphs, excellent graph readability, and supports recursive nesting of subgraphs. The latter point is mandatory for a hierarchical representation of complex software structures.



```
routine: _memcpy
char *s;
int  l;
     while (l--) *d++ = *s++;
<empty>
<empty>
counter ++;
MOV DPP0,#PAG _counter
MOV R8,[R0+]
MOV R12,POF _counter
ADD R12,#01h
MOV POF _counter,R12
RETS
```

```
routine: _strcmp_x
for ( ; *s == *t; s++, t++ , counter++)
if ( *s == '\0' )
return( 0 ) ;
<empty>
<empty>
return( *s - *t ) ;
```

```
routine: _Func1
CapitalLetter   CharPar2;
  REG CapitalLetter        CharLoc1;
  REG CapitalLetter        CharLoc2;
  CharLoc1 = CharPar1;
  CharLoc2 = CharLoc1;
  if (CharLoc2 != CharPar2){
counter+=91;
return (Ident1);
           }
else {
        counter+=93;
        return (Ident2);
}
```

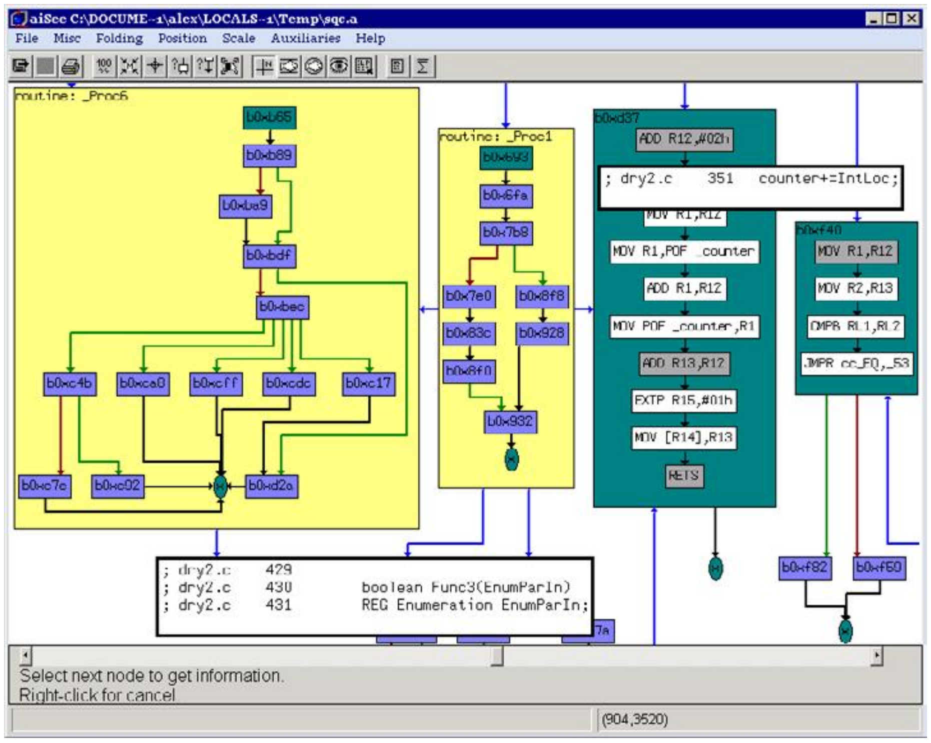
Fig. 2. Control flow subgraphs: Assembly view and C source code view

3 Hierarchy Description

The top-level graph is the call graph. It shows the calling relationship between procedures, represented by nodes. An edge represents a call to a procedure (see Figure 1). External procedures are shown in a different color, enabling the interface to modules to be easily identified. The user can display a color legend together with the call graph. The graph colors are fully customizable.

Using subgraph nesting operations in the graph browser enables the content of a procedure, i.e. the procedure's control flow graph to be visualized. The nodes of the control flow graph represent basic blocks (straight line sequences of assembly instructions) and edges represent the control flow within the procedure (see Figure 2). In the C Source View, the basic blocks are labeled with C source code snippets. Again, the subgraph nesting operations enable the user to view the content of a basic block, i.e. the sequence of assembly instructions corresponding to the particular C source code snippet.

The subgraph nesting operations enable interwoven call control-flow graphs to be explored, thus allowing the programmer to concentrate on certain parts of the graph and explicitly analyze specific points of interest. Additional node tooltips enable viewing source code comments, internal addresses and source file destinations (see Figure 3).



```
; dry2.c    351   counter+=IntLoc;
```

```
; dry2.c    429
; dry2.c    430          boolean Func3(EnumParIn)
; dry2.c    431          REG Enumeration EnumParIn;
```

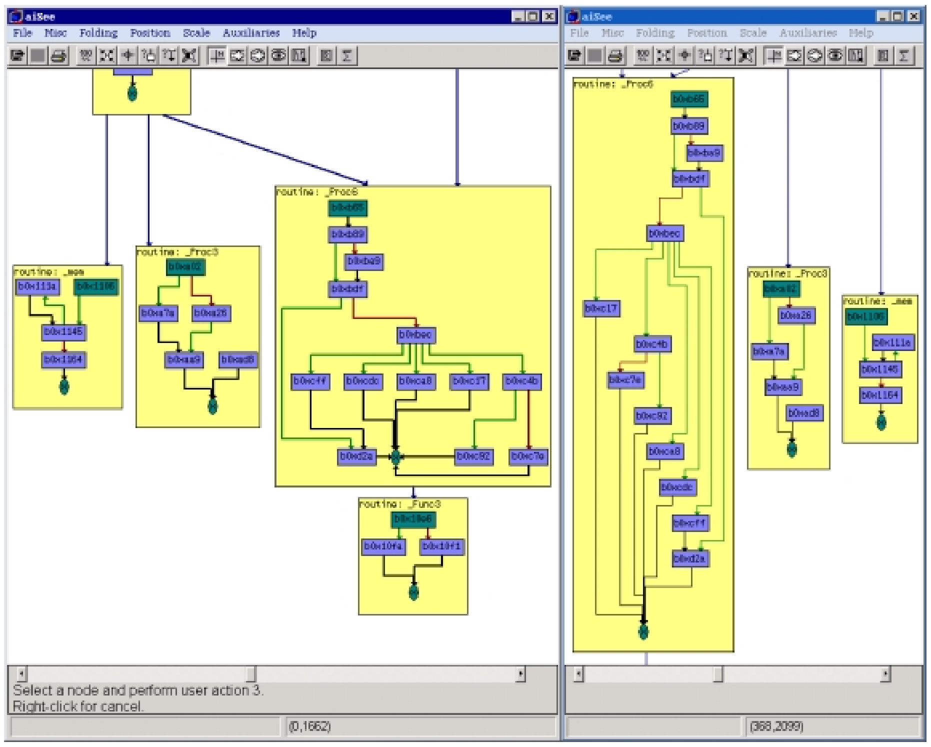
Select next node to get information.
Right-click for cancel

Fig. 3. Interwoven call graph and control flow graph with node tooltips

4 Navigation

The graph browser supports scaling of graphs and provides several features for navigating through graphs. The features include centering arbitrary nodes and setting the focus on one procedure, while hiding the surrounding call graph. Furthermore, the user can interactively explore a path through the graph by means of following edges.

aiSee features various layout algorithms and a number of customizable layout parameters like graph orientation, distance between nodes, edge styles, etc. (See Figure 4.) The animation features (smooth transitions) allow the user to keep track of the changes induced by relayout or subgraph nesting operations.



Select a node and perform user action 3.
Right-click for cancel.

Fig. 4. Normal layout and linearized layout. In the linearized layout, the vertical order of basic blocks corresponds to their position in the C source file. Thus, the linearized layout algorithm provides for an even better view of initial C source code structures like loops and conditional jumps.

When working with huge graphs, zooming on a focus can result in losing sight of important contextual information. Therefore, aiCall provides an overview window (panner) and sophisticated graph distortion techniques (polar and cartesian fish-eye views [Abs,San91]) in order to improve working with large graphs.

The panner offers a scaled-down view of the entire graph in a separate window, whereas fish-eye views imitate the well-known fish-eye lens effect by magnifying the focus area and displaying other parts of the graph with less detail. The parts of the graph that are further away from the focus appear slightly squashed, meaning the further nodes are positioned away from the focus, the smaller they appear in the graph window. Thus, the user can concentrate on areas of particular interest yet being able to consider their context and even to overview the entire graph (see Figure 5).

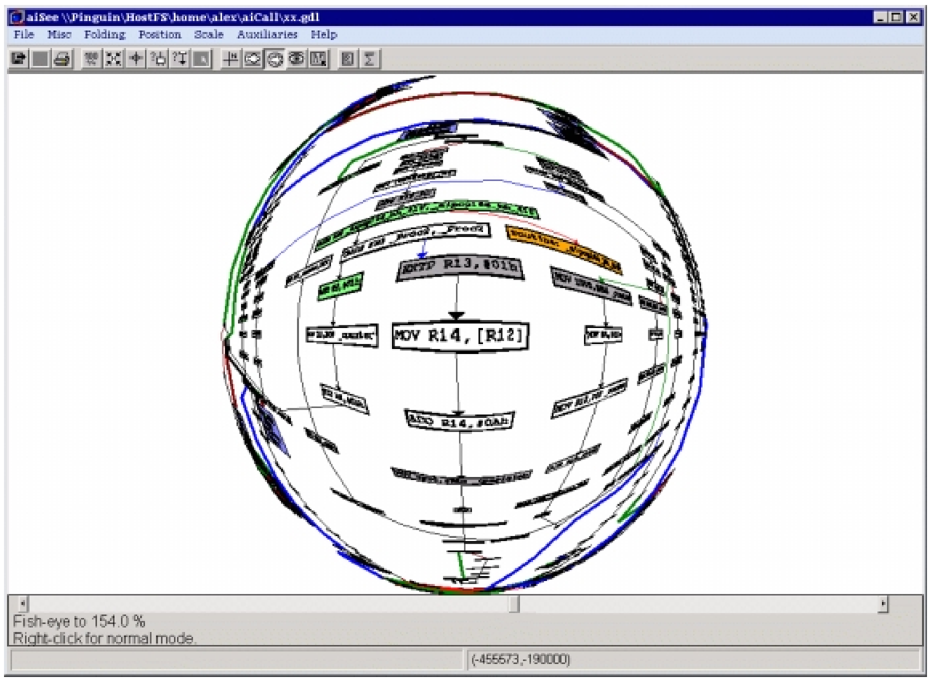

aiSee \\Pinguin\HostFS\home\alex\aiCall\xx.gdl
File Misc Folding Position Scale Auxiliaries Help
MOV R14, [R12]
Fish-eye to 154.0 %
Right-click for normal mode.
(-455573,-190000)

Fig. 5. Polar fish-eye view

5 Program Documentation

aiCall integrates a complete framework for printing graphs, export of graphs in colored Postscript format (on multiple pages for large graphs) and other picture formats. This permits more concise program documentation produced faster and understood more easily.

6 Compatibility with Code Optimization

aiCall is compatible with the Code Compaction Suite aiPop [Fer01,FH01b]. aiPop automatically reduces code size and improves the code quality of assembly files produced by the Tasking C compiler for C16x/ST10 (see Figure 6). The optimized files can be visualized with aiCall. aiCall also supports viewing the call and control flow graphs of both original and optimized files in the same graph window. aiCall fully supports the aiPop color coding scheme. Additional node tooltips enable viewing detailed information about the optimizations performed by aiPop.

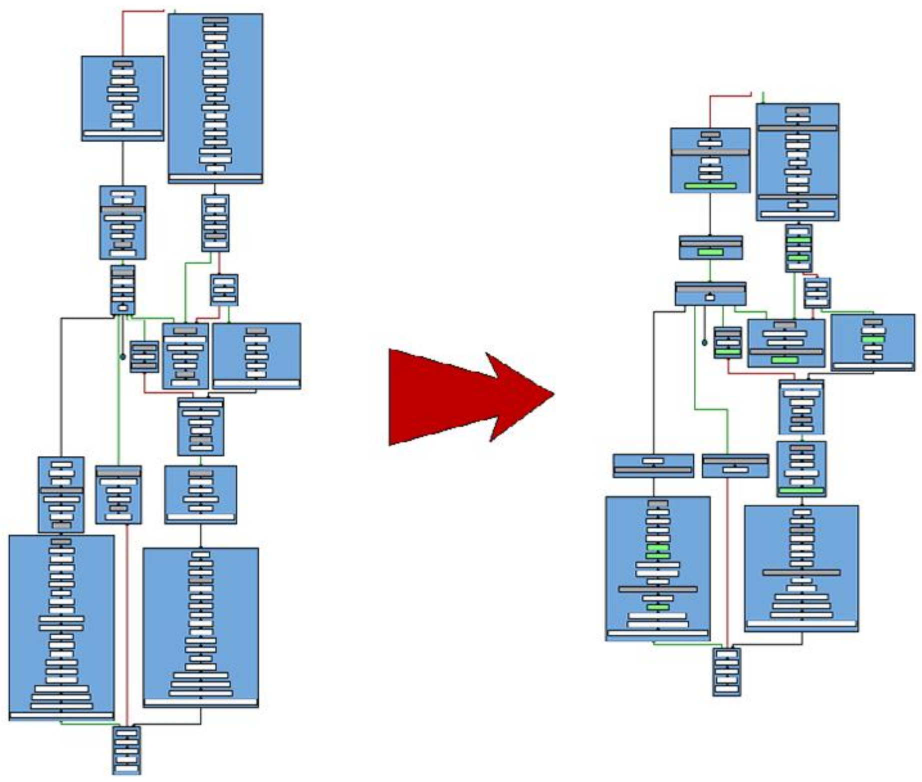


Fig. 6. Control flow graph of a C16x/ST10 application before and after optimization with aiPop

7 Integrating Stack Usage Analyses

In typical embedded systems programming, stack memory has to be allocated statically by the programmer. Underestimating stack usage can lead to serious runtime errors which can be difficult to find. Overestimating stack usage means a waste of memory resources. With an optional Stack Usage Analysis Module, aiCall provides automatic tool support to calculate the stack usage of embedded applications for C16x/ST10.

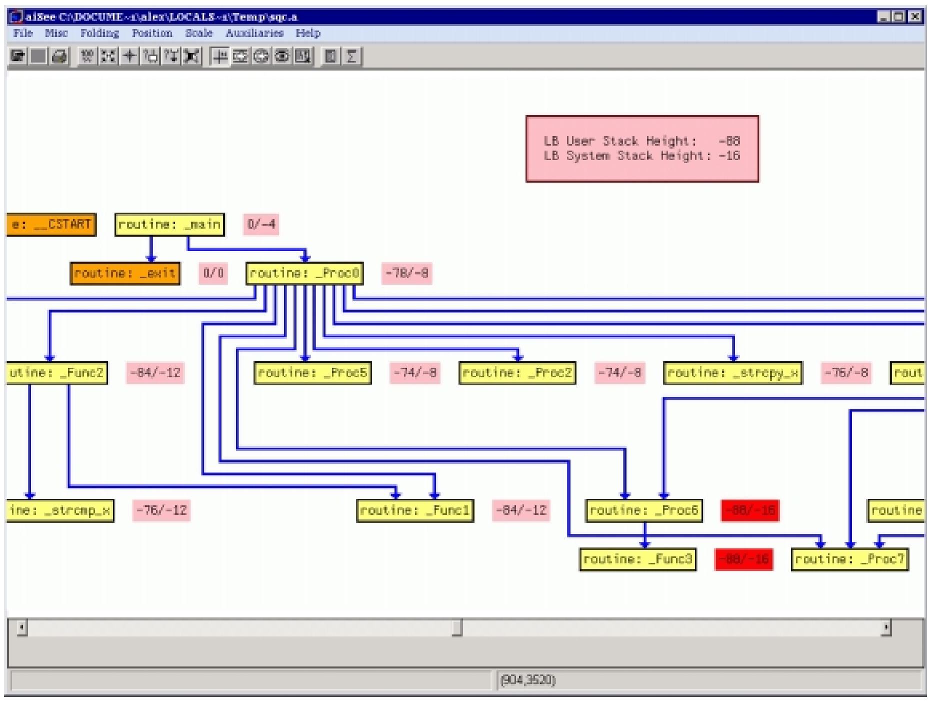


Fig. 7. Call graph with stack usage annotations

Stack height differences are shown as annotations in the call and control flow graphs (see Figure 7). Critical program parts can be easily recognized thanks to color coding. Additional node tooltips enable viewing detailed context information about the stack usage of particular instructions, basic blocks, and procedures. The user can interactively select multiple entry points for the stack usage analysis (see Figure 8).

With the Stack Usage Analysis Module, aiCall not only reduces development effort but also helps to prevent runtime errors due to stack overflow. Furthermore, the analysis results provide valuable feedback in optimizing the stack usage of embedded applications [FH01a].

8 Summary

aiCall enables a visualization of programs going far beyond standard textual representation, thereby enhancing productivity, as complex program code can be understood faster. aiCall is based on the graph layout software aiSee, formerly known as VCG [San91], winner of the graph layout competitions in Princeton, New Jersey, USA, in 1994 [TT94] and in Passau, Germany, in 1995 [Bra95]. aiSee provides for excellent graph readability, fast layout calculation even for very large graphs, and a rich collection of features.

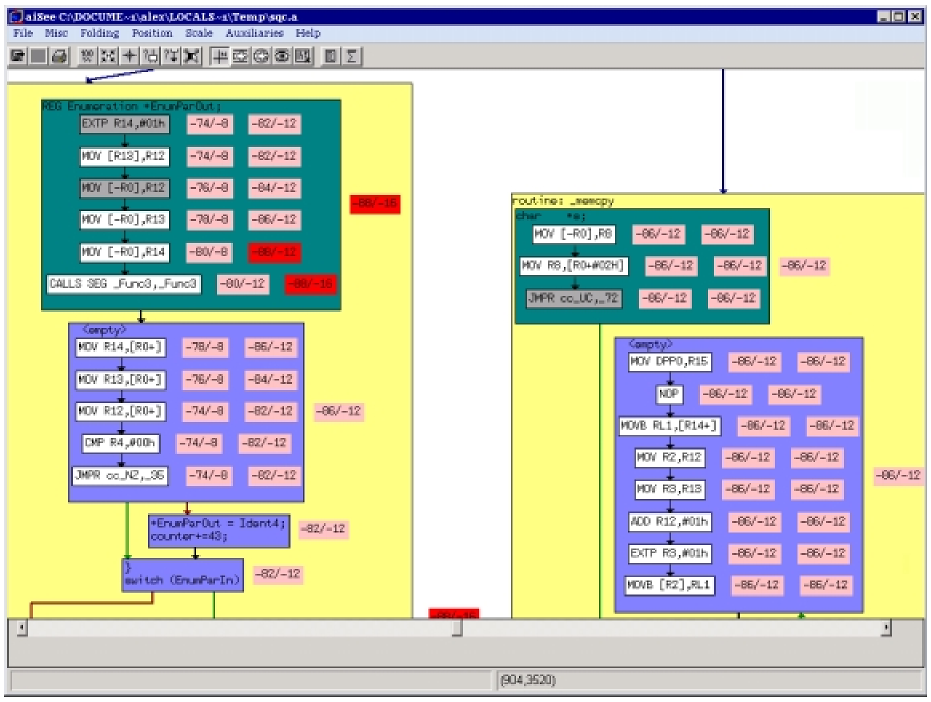


Fig. 8. Control flow subgraphs with stack usage annotations

References

[Abs] AbsInt Angewandte Informatik GmbH. *aiSee Graph Visualization: User Documentation – Windows Version.* http://www.aisee.com/manual/windows.

[aGVS] aiSee Graph Visualization Software. *aiSee Home Page.* http://www.aisee.com.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Bra95] F. Brandenburg, editor. *Symposium on Graph Drawing GD'95; Passau, Germany, September 20-22, 1995. Proceedings*. Lecture Notes in Computer Science. Springer, 1995.

[EB] Alexander Evstiougov-Babaev. *Graph Description Language in a Nutshell*. http://www.aisee.com/gdl/nutshell.

[EBF01] Alexander Evstiougov-Babaev and Christian Ferdinand. Program Visualization Support for C16x. *CONTACT, Infineon Technologies Development Tool Partners Magazine*, 3(9):64–65, February 2001.

[Fer01] Christian Ferdinand. Post Pass Code Compaction at the Assembly Level for C16x. *CONTACT, Infineon Technologies Development Tool Partners Magazine*, 3(9):35–36, February 2001.

[FH01a] Christian Ferdinand and Reinhold Heckmann. Ein Bild sagt mehr als tausend Worte. *Design&Elektronik*, 8, August 2001.

[FH01b] Christian Ferdinand and Reinhold Heckmann. Liebling, ich habe den Code geschrumpft. *Design&Elektronik*, 1:113 – 115, January 2001.

[Inf97] Infineon. *Instruction Set Manual for the C16x Family of Siemens 16-Bit CMOS Single-Chip Microcontrollers*, 1997.

[Lan99] Marc Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, TFB 14, Saarland University, November 1999.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[San91] Georg Sander. *Visualisierungstechniken für den Compilerbau*. PhD Thesis, Lecture Notes in Computer Science. Pirrot, 1991.

[TT94] R. Tamassia and I.G. Tollis, editors. *DIMACS International Workshop, GD'94, Princeton, New Jersey, USA, October 10 - 12, 1994. Proceedings*. Lecture Notes in Computer Science. Springer, 1994.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.

Chapter 5
Future Perspectives

Introduction

Stephan Diehl

Saarland University,
FR 6.2 Informatik,
PO Box 15 11 50,
D-66041 Saarbrücken, Germany,
diehl@cs.uni-sb.de

It has been a long way since Knowlton's movie about list processing with the programming language L6 [14]. Thousands of algorithm animations, hundreds of systems, and numerous case studies and evaluations have been produced since. But don't get me wrong, it's not all said and done? By and large software visualization research has concentrated on a few aspects of software. So you might ask, *what should it concentrate on in the future?* To answer this question we present a quantitative map of existing research and discuss some cross-topic research themes.

There exist several taxonomies for software visualization in the literature [20, 22,23,25]. We propose another one here for the sole purpose of identifying almost unexplored research areas[1]. The taxonomy is based on *what* and not *why* or *how* it is visualized.

Certainly there are various dimensions which could be used to categorize research in software visualization. In Figure 1 we use two dimensions: the classical abstraction layers of a software system (hardware, virtual/abstract machine, program and system) and the static and dynamic phenomena of these layers. The map is incomplete in the sense that one could add additional layers (e.g. operating system) or structures (e.g. project structure). In addition to the qualitative information we use shades of gray to indicate how much published research exists in certain areas of software visualization. Dark gray indicates a high number (more than 100), medium gray a low number (more than 10) and light grey almost no published research. The numbers are based on web searches using two comprehensive search engines, namely CiteSeer (www.citeseer.com) and Google (www.google.com). For the search we used the abstraction layers as keywords

[1] Our taxonomy is closest to Myers' taxonomy [20]. He proposes a 2×3 matrix ({dynamic, static} $\times$ {code, data, algorithm}), but provides no quantitative analysis.

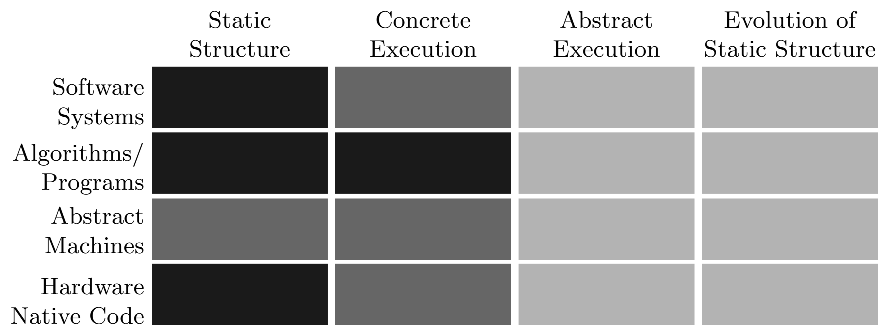


Fig. 1. A map of software visualization research

together with “visualization” or “software visualization”. E.g. for “algorithm visualization”, CiteSeer found 22 papers with these keywords in their titles, for “program visualization” it found more than 300. Then we looked at the titles, abstracts or contents to decide which aspect is covered by a paper. For the combination “abstract machine” and “visualization”, CiteSeer found no entries. Using Google and browsing through 845 web pages which contained these keywords, we found that there are more than 10 papers which address the topic of visualizing abstract machines. As the same concepts go by different names in different communities this study does not claim to be complete or comprehensive in any way, but we think that it gives a rough orientation on the activity of research in these areas.

1 Discussion and Examples

We will now discuss some of the regions in this research map and give examples of prototypical or seminal work in the more unexplored regions including references to articles in this volume.

Static Structure. The static structure of programs and systems has been visualized in various ways including pretty printing, control-flow diagrams, and UML diagrams. The reader certainly will have seen many of those before. Korhonen et. al. [15] suggest to have students run a program once or more with different input data and indicate the path coverage of those input data in the control flow diagram.

Concrete Execution. The execution of (small) programs has been visualized in various ways and many animations are accessible on the internet. The blurry distinction of program and algorithm animations based on the level of abstraction used in the visualization is widely accepted. Visualization systems to automate the production of such animations have been created. They mostly differ in what kinds of data structures and algorithms are visualized, and what programming languages are used. Program animations have been mainly used in education.

Abstract machines provide an intermediate language stage for compilation. They bridge the gap between the high level of a programming language and the low level of a real machine. The instructions of an abstract machine are tailored to the particular operations required to implement operations of a specific source language or class of source languages [7]. Visualizations of a few abstract machines exist [10,18] and a web-based generator for interactive animations of abstract machines called GANIMAM [8] has been developed as tool to teach and design abstract machines.

Abstract Execution. Most work in algorithm animation addresses the visualization of the execution of programs with concrete input data. The visual execution of programs with abstract input data, i.e. representations which only reflect the relevant properties, has hardly ever been investigated. Michail [19] uses abstract states in form of partial trees, which represent a possibly infinite number of trees, for visual programming of binary tree algorithms. There is a rich theory of abstract interpretation as a basis for program analysis [21] and first results on how to use this for visualization of programs [28,5].

Evolution. In the real world software systems are not designed and implemented once and forever, but evolve over time. Metric data gathered during such an evolutionary process has been visualized using typical information visualization methods [1]. Configuration management systems are widely used to record and control the evolution of software systems. VRCE [9] and WinCVS [26] extend such configuration management systems by the ability to draw the version graph of a single file.

2 Cross-Topic Research Themes

The increasing importance of software visualization in software engineering and in particular in re-engineering is emphasized by two recent studies [16,17,3]. These studies reveal many weaknesses of existing tools. Koschke interviewed more than 100 researchers in the areas of software maintenance, reverse engineering and re-engineering. About 40% feel that software visualization is absolutely necessary for their research, for another 42% software visualization is important but not critical.

Typical software issues like portability, scalability and maintainability are certainly appropriate for software visualization systems as well, but we feel that there are more genuine issues in software visualization. As noted by many authors before, there is no such thing as the one right visualization technique, but we might find ourselves in an even worse situation enhancing visualization methods which are not useful at all because we visualize the wrong aspects of software.

Virtual Reality. Knight and Munro[13] evangelize 3D for software visualization. Instead of visualizing graphs as nodes connected by lines, in their *Software World* visualization [12] they use a buildings or even cities metaphor; nodes are rooms or buildings and edges are doors, aisles or streets. As a consequence natural navigation through the graph is along these edges. Their work is technology driven focussing on how and not what is visualized.

Distributed Applications. With the boost of the world-wide-web the trend to distributed software systems has been reinforced. As Scott McNeally, CEO SUN Microsystems, said: "The net is the computer". It is characterized by applications with high heterogeneity. For example, a single application can employ different programming languages, operating systems, protocols, security mechanisms, and component technologies. Even worse, the structure of an application can change at runtime depending on what resources are currently available on the net. Communicating software agents belong to this kind of architectures. Van Lengen and Bähr describe visual and interactive components allowing the introspection and manipulation of such open applications [27].

Metaphors. Metaphors are commonplace in computer science: machines and automata, tapes, states, nodes and edges, records, files, windows, just to name a few. For example, a Turing machine is a mathematical model comprised of sets, functions, and/or relations. The *machine* analogy lets us transport aspects from the physical world to the mathematical and thus helps to better understand the mathematical model. We might even think of gear wheels and how one drives the others, once we start to turn one of them. A metaphor evokes a mental image[2]. There is much evidence that we think in images and that in a last step we verbalize our thoughts. Famous scientists like Kepler, Kekule[3] and Einstein reported that their scientific thinking started with images and at the very last they tried to cast their results into formula. Mental images need not be visual, but can be in any sense: hearing, touch, smell, or even mood. The goal of software visualization is not to produce neat computer images, but computer images which evoke mental images for comprehending software better. Finding new metaphors thus will not just produce better visualizations, but it will also improve the way we talk about systems. Fishwick [11] suggests visual metaphors based on 3D computer graphics. Baloian and Luther [2] argue that software can be visualized without producing computer images, but using other perception channels to create mental images.

Visualization Pipeline. The creation of computer images is just the last step in the visualization pipeline: data gathering, data analysis, and visualization. In scientific visualization research is performed on all three phases. For software visualization use and development of analysis methods for focussing visualizations have been neglected.

[2] Given the non-conclusiveness of psychological evidence for what mental images are [24] the current author dares to share his metacognitive view here [4].

[3] He reported that a vivid dream of a snake eating its tail gave him the idea of the structure of the Benzene Ring.

System Visualization. Although software visualization is about software, we have to realize that software is used in real world contexts. Thus we might define a system as the combination of software, hardware (devices) and humans (users). To support understanding and design of such systems, e.g. of work flows in such a system, we have to visualize more than just the structure or execution of the software. In addition the activity, which happens in the real world, must be tracked or simulated.

3 Steering into the Future

"My interest is in the future because I am going to spend the rest of my life there. "

– *C.F. Kettering*

So what are the challenges in software visualization research and what are the next steps to be taken to increase its impact? The following comments are based on discussions with and suggestions of the authors of this volume.

Breaking New Ground. Considering the many unexplored research areas and the discussion above, the answer to our initial question might be that *software visualization research should not concentrate on a certain topic in the future.* We expect that exploring all aspects at all layers of software will lead to synergies and thus will ultimately stir the traditional areas of software visualization as well.

Integration. Software visualization will be doomed to stay an academic endeavor, if we do not succeed to integrate it into working environments and thus into the work flow of programmers, designers and project managers. To facilitate such integration existing standards must be adopted or extended, and if needed we must agree upon new standards.

Theory. Certainly the effectiveness of software visualizations for all kinds of applications must be evaluated. Based on such empiric data cognitive and pedagogical theories can be formulated and validated. Ultimately, these should guide the design and use of future software visualization systems.

Forum. As there is still no established journal, conference or workshop, many research groups have been unaware of previous work and reinvented the wheel. The software visualization community needs a forum to share best practices and to promote the state of the art.

4 Conclusion

The papers in this volume give a snap shot of the current state of the art of software visualization. All papers in this final chapter have two things in common: they present research in those cross-topic themes discussed above and they are off the beaten path and open the door for others to follow.

References

1. M.J. Baker and S.G. Eick. Visualizing software systems. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy; May 16-21, 1994). IEEE Computer Society Press, 1994.
2. Nelson Baloian and Wolfram Luther. Visualization for the Mind's Eye. In *[6], 2002*.
3. Sarita Bassil and Rudolf K. Keller. Software Visualization Tools: Survey and Analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC2001) (to appear)*, Toronto, Ontario, Canada, 2001.
4. Alan F. Blackwell. Metacognitive Theories of Visual Programming: What do we think we are doing? In *Proceedings of IEEE Symposium on Visual Languages VL96*. 1996.
5. B. Braune and R. Wilhelm. Focussing in algorithm explanation. *Transactions on Visualization and Computer Graphics*, 6(1), 2000.
6. Stephan Diehl, editor. *Software Visualization*, volume 2269 of *LNCS State-of-the-Art Survey*. Springer Verlag, 2002.
7. Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract Machines for Programming Language Implementation. *Future Generation Computer Systems*, 16(7), 2000.
8. Stephan Diehl and Thomas Kunze. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems*, 16(7), 2000.
9. DuraSoft GmbH. RCE, VRCE, BDE. `http://wwwipd.ira.uka.de/~RCE`.
10. Manfred Hauswirth, Mehdi Jazayeri, and Alexander Winzer. A java-based environment for teaching programming language concepts. In *Proeceedings of ASEE/IEEE Frontiers in Education '98 Conference, Tempe, AZ*, 1998.
11. John F. Hopkins and Paul A. Fishwick. THE rube (tm) METHODOLOGY FOR 3-D SOFTWARE. In *[6], 2002*.
12. C. Knight and M. Munro. Comprehension with[in] virtual environment viusalizations. In *Proceedings of the IEEE 7th International Workshop on Program Comprehension*, 1999.
13. C. Knight and M. Munro. Visualising software – a key research area (short paper). In *Proceedings of the IEEE International Conference on Software Maintainance*, 1999.
14. K. Knowlton. L6: Bell Telephone Laboratories Low-Level Linked List Language. 16-minute black-and-white film, 1966.
15. Ari Korhonen, Erkki Sutinen, and Jorma Tarhio. Understanding Algorithms by Means of Visualized Path Testing. In *[6], 2002*.
16. Rainer Koschke. Software Visualization for Reverse Engineering. In *[6], 2002*.
17. Rainer Koschke. A Survey on Software Visualization for Software Maintenance, Re-Engineering and Reverse Engineering. `www.informatik.uni-stuttgart.de/ifi/ps/rainer/softviz`, 2001.
18. J. García Martín and J.J. Moreno Navarro. Visualization as debugging: Understanding/debugging the warren abstract machine. In *1st International Workshop on Automated and Algorithmic Debugging, Linkøping (Sweden)*. Lecture Notes in Computer Science, Springer Verlag, 749, 1993.
19. Amir Michail. Teaching binary tree algorithms through visual programming. In *Proceedings of IEEE Symposium on Visual Languages*, 1996.

20. B. Myers. Taxonomies of visual programming and program visualisation. *Journal of Visual Languages and Computing*, 1, 1990.
21. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
22. M. Oudshoorn, H. Widjaja, and S. Ellershaw. Aspects and taxonomy of program visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*. World Scientific Press, Singapore, 1996.
23. B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), 1992.
24. Zenon W. Pylyshyn. Mental Imagery: In search of a theory. *Behavioral and Brain Sciences*, (to appear), 2002.
25. G.C. Roman and K.C. Roman. A taxonomy of program visualization systems. *Computer*, December 1993.
26. Strata Inc. WinCVS Homepage. `http://www.cvsgui.org`.
27. Rolf Hendrik van Lengen and Jan-Thies Bähr. Visualisation and Debugging of Decentralised Information Ecosystems. In *[6], 2002*.
28. Reinhard Wilhelm, Tomasz Müldner, and Raimund Seidel. Algorithm Explanation: Visualizing Abstract States and Invariants. In *[6], 2002*.

Visualization for the Mind's Eye

Nelson Baloian and Wolfram Luther

Institut für Informatik und Interaktive Systeme der GMU Duisburg,
Lotharstraße 65, 47057 Duisburg, Germany
{Baloian, Luther}@informatik.uni-duisburg.de

Abstract.

Software visualization has been almost exclusively tackled from the visual point of view; this means visualization occurs exclusively through the visual channel. This approach has its limitations. Considering previous work for blind people we propose that complementing usual approaches with those techniques used to develop interfaces for non-sighted people can enhance user awareness of logical structures or data types using different perception channels. To achieve better comprehension, we deal with new or augmented interfaces built on top of standard systems for data visualization and algorithm animation. The notion of specific concept keyboards is introduced. As a consequence, modern information and learning systems can be designed in such a way that not only sighted but also blind users can navigate within these systems.

1 Value and Problems of Software Visualization

Software visualization deals with the animation of algorithms, including numerical, geometric, graphic, and graph algorithms, as well as the visualization of data structures in information systems or in the computer's memory while certain complex processes are performed. One of its main goals is to achieve a better understanding of complex programs, processes, and data structures by means of showing complex digitized images displayed on a CRT- or LCD-monitor or a printing device. Through these images transmitted by the visual channel, users should generate planar or spatial structures with dynamic objects in their minds.

The visual channel permits a rapid overview of structures after an adequate abstraction process, a separation of important objects from less important ones whenever the former are distinguishable by graphic attributes, and a real time processing of dynamic process data output - furthermore, a strong data compression in the mind accompanies the high-band wide reception process. However, the visualization approach has several remarkable consequences: Multidimensional structures are projected to a plane which results in numerous design problems and higher data structures; parallel processes are serialized leading to nested screens or dialogues. Here, we will not contribute to the 'Layout Creation and Adjustment Problem' for graphs in visualization and animation systems [9].

Too often, the resulting graphic outcome is highly complex if we consider rasterized animated sequences and is strongly compressed during the reception process. Only a few details remain in memory. This fact is widely discussed in the literature. J. Norseen [18] deals with the conversion of 2D retinal visual sensory information signals into 3D semiotic mental representations and describes a visual semiotic language built from a finite alphabet of basic images: In constructing mental images certain left occipital areas of the human brain seem to perform the same neurological functions while the so-called Broca-Wernicke area does the conversion of aural sensory signals into a finite set of sound types producing neurolinguistic patterns. Norseen suggests that only about forty sound types and thirty images form the basis of expressed aural and visual languages.

Consequently, it would seem to be of interest to bypass, or better still, to complement standard graphic displays and search for alternative ways to provide logical structures using different perception channels to human minds provoking equivalent or alternative impressions and images.

The paper begins with a discussion of parallel reception modes and introduces the notion of concept keyboards. Then different ways for implementing an enhanced perception are discussed; this is followed by a case study. We formulate some ideas on the automatic generation of concept keyboards, look at recent work in the area and finish with some conclusions.

2 Parallel Reception Modes

Our working hypothesis will be that not only sighted but also blind users should be able to use and navigate within systems implementing new or augmented interfaces using enhanced perception tools to achieve software visualization for the mind's eye [16]. Thus, we do not intend to develop a system for blind people only, on the contrary, we claim that by proposing complementary perception channels and navigating facilities there will be a real enhancement for "normal users" when systems are navigable and usable for people with disabilities. It is reported in [17] that blind people develop special forms of navigating within an unknown environment and represent spatial structures with cognitive difficulty. This is true not only for the real world but also in virtual computer based environments. Certain evidence of this is given by experiences done according to the HOMER UIMS approach by Savidis and Stephanidis [23]; this approach consists of developing dual user interfaces for integrating blind and sighted people. To achieve this goal standard visualization elements like control element icons, tool menus, short cuts, logical structures with nodes and links, hypertext, images and animated sequences are enriched with acoustic elements or haptic interfaces, which allow direct interaction of the user with objects of the model used for the computer to represent the problem being explained or presented. To navigate independently from the graphic output we introduce the idea of a draft keyboard, which is realized by redefining keys on a traditional keyboard, by a matrix of small keys on a graphic tablet or by mapping them with the help of problem-specific hardware.

3 Ways to an Enhanced Perception

At present, virtual environments are basically built on visual displays, with some use of auditory and very little haptic information. The International Community for Auditory Display (ICAD) is a forum for presenting research on the use of sound to display data, monitor systems providing enhanced user interfaces for computers and virtual reality systems [12]. Research areas include the auditory exploration of data via sonification and audification, perceptual issues in Auditory Display systems and sound in immersive interfaces and virtual environments.

Mapping scientific data redundantly to visual and aural elements may increase the perception of the information and can lead to better insight and understanding. Conveying the same information using different channels to transmit it to the user becomes an important design element in systems where the network bandwidth is limited.

To complement the visual channel we have to design an acoustic interface which delivers at least the same information as the graphic one. So we have to develop a correspondence between visual and aural control elements as well as acoustic and graphic attributes. An earcon characterized by a typical melody can be added to any iconic control object. Earcons are abstract musical tones that can be used in structured combinations to create auditory messages. In 1989 Meera Blattner [7] introduced earcons as nonverbal audio messages to provide information to the user about computer objects, operations or interactions. Earcons are constructed from simple melodies also called motifs. A motif is a typical tonal pattern sufficiently distinct to represent an individual recognizable entity. The most important features of motifs are instrument, rhythm, and pitch. Earcons for such operations as ‘Play’, ‘Go left’, ‘Go right’, ‘Forward’, ‘Back’, ‘Jump’, ‘Start’ and ‘Stop’ could be created. It is possible to produce higher level earcons such as ‘Next problem’ or ‘Close program’ [8] and to create hierarchical structures. For modeling objects in our virtual world the following correspondences are possible:

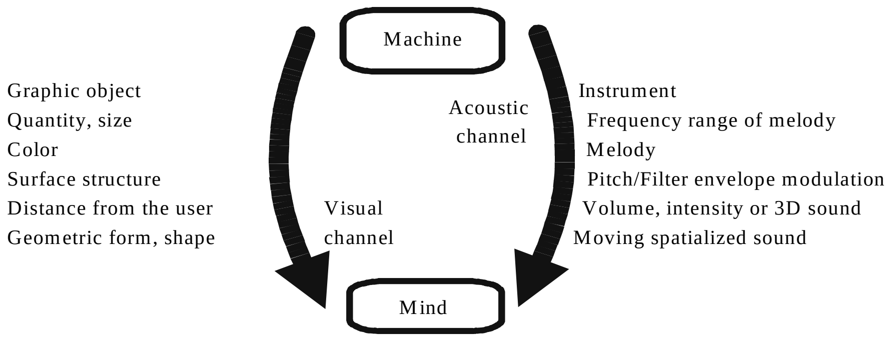


Fig. 1. Correspondences between graphic and aural attributes

According to Bissell [6] people’s association between tonal pitch and spatial position depends on the cultural context. Moreover, the correspondences proposed in Figure 1 do not fully match the established psycho-physiological basis for Western culture. A tone, being of definite pitch, is the type of sound particularly relevant here. Tones represent a striking metaphor of location and motion, a change in pitch is perceived as being analogous to actual spatial motion and as taking place along the vertical dimension. Tonal brightness is associated with visual brightness, which is associated with visual or physical highness in space.

Nodes and links are the constituent components of Hypertext. While nodes can easily be complemented by text explaining the content in a straightforward way, there exist several possibilities for transforming links into sound. Neighboring nodes can be enriched by text hints or a pair of sounds; – a path can be illustrated by sequences of sounds. All these representations are volatile and must be activated after some time. This can also be done by using hotspots as particular restricted areas of larger graphics. Thus, moving into such an area or leaving it would trigger auto-narration or special wave files.

An interesting aspect of modern graphic user interfaces is they offer an easy and comfortable way to navigate and interact with software systems directly on visualized structures or through control elements like buttons, scrollbars, and dialogues.

- Users point directly at displayed interactive objects.
- They traverse the graphic representation of a logical or a hierarchical data structure with the aid of arrow keys or a pointing device.
- They manipulate or search for interactive objects by exploring a matrix of small areas called a concept keyboard with the aid of arrow keys or a pointing device and graphic tablet.

However, combining visual output with control elements is not appropriate for a blind person. Whereas the parallel use of icons and earcons enables blind people to control the system, the navigation on visual representations of internal data structures should be accompanied by appropriate concept keyboards custom-built for the application.

Currently, there are industries manufacturing touch devices called Concept Keyboards which can be connected to the computer through a serial port. For example, a touch pad consists of a flat touch-sensitive polycarbonate surface (A2, A3 or A4 size) made up of 128 or 256 cells set out in a rectangular array and allows the user to select the keyboard layout best-suited to the required application. Up to 256 programmable keys, defined individually or in groups with pre-designed overlays and blank templates can be customized using one of the existing overlay designer software packages. An overlay can have many different layers of information, pictures, video or sound. Paper sheets can be inserted on which objects are embossed and words are written in Braille. A blind person controls an application by pressing on the embossed pictures or Braille words. When particular areas of the concept keyboard are pressed, a digitized sound can be heard. A number of tactile overlays have been designed to be placed on the concept keyboard. Users can move their fingers along the tactile path, and, when they press certain ’nodes’, the computer executes appropriate control commands or plays sound sequences [1].

4 Case Studies for Enhanced Perception

4.1 Visualization of Algorithms in Computer Graphics and Image Processing

ViACoBi [10, 11, 15] is interactive multimedia courseware visualizing Computer graphics and Image processing algorithms; it provides a new learning environment for undergraduate students to accompany classical lecture and lab courses. It proposes eight lessons, each one providing a certain interactive presentation mode with links to relevant technical facts and terms in a glossary and a context-sensitive help function. It also includes exercises, and a visualization and animation toolbox to construct relevant artifacts and to test algorithms in different scales and with various input parameters in a stepwise execution mode displaying all relevant variable contents. One aim of the system is the visualization of rastering algorithms to digitize straight lines, circles and ellipses.

We explain our approach with an example concerning 4-way and 8-way stepping in digital geometry. In raster graphics, there are four horizontal and vertical neighbors (x±1,y) and (x,y±1) of a pixel (x,y) or eight neighbors if we add the four diagonal neighbors (x+1,y±1) and (x-1,y±1). By stepping via 4- or 8- adjacent points we obtain 4- or 8-paths or so-called raster curves built from adjacent 4- or 8-neighbors. A concept keyboard should support 4-way or 8-way stepping from one position to the adjacent points in any of the 4 or 8 directions, respectively. This can be done via 4 or 8 arrow-keys which allow navigating stepwise on paths and shapes.

The keys are labeled with the directions east coded by the number 0, northeast by 1, north by 2 and so on. Each path can be described by a starting point (x_0,y_0) which can be reached by the focus-key and a sequence of directions, called the chain-code. There is an analogue three dimensional extension when we consider octrees and voxels with 26 neighbors. Here, we can use the layers of a 2D stepping keyboard. Starting from a *RasterCurve* class, its parameters and methods, and the sequence of method calls, it is possible to generate and to design the concept keyboard automatically. (This will be addressed later in point 5 of this paper.)

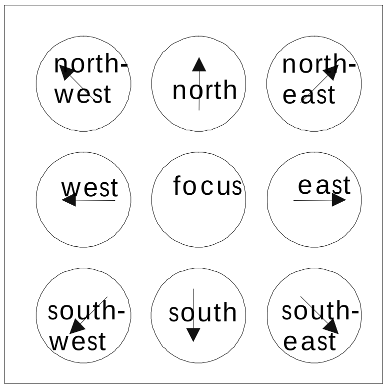
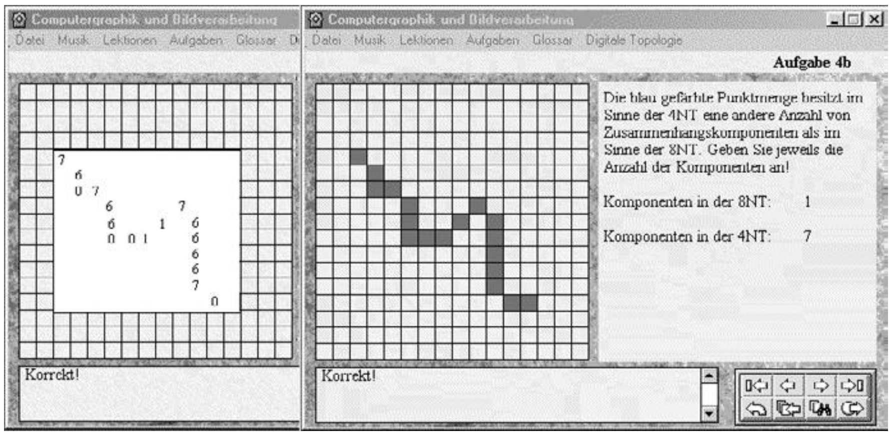


Fig. 2. Concept keyboard for 4-way or 8-way stepping

In the exercise part (see Figure 3) we ask for the number of the connected components of an 8-path in the case of the four neighbors topology where only horizontal and vertical steps coded by even numbers are allowed. It is more intuitive to use the aural representation and directly count the odd numbers within the chain-code enumeration than to derive the result from the visual representation of the path in the object area, which contains much more irrelevant information. Furthermore, the blind user can directly construct or modify raster curves such as lines, circles or ellipses by using the concept keyboard. The key in the center can be used to skip to the beginning of the curve. Correct and incorrect steps are signaled by different acoustic messages.


Computergraphik und Bildverarbeitung
Datei Musik Lektionen Aufgaben Glossar Digitale Topologie
7 6 0 7 6 7 6 1 6 0 0 1 6 6 6 7 0

Aufgabe 4b

Die blau gefärbte Punktmenge besitzt im Sinne der 4NT eine andere Anzahl von Zusammenhangskomponenten als im Sinne der 8NT. Geben Sie jeweils die Anzahl der Komponenten an!

Komponenten in der 8NT: 1

Komponenten in der 4NT: 7

Korrekt!

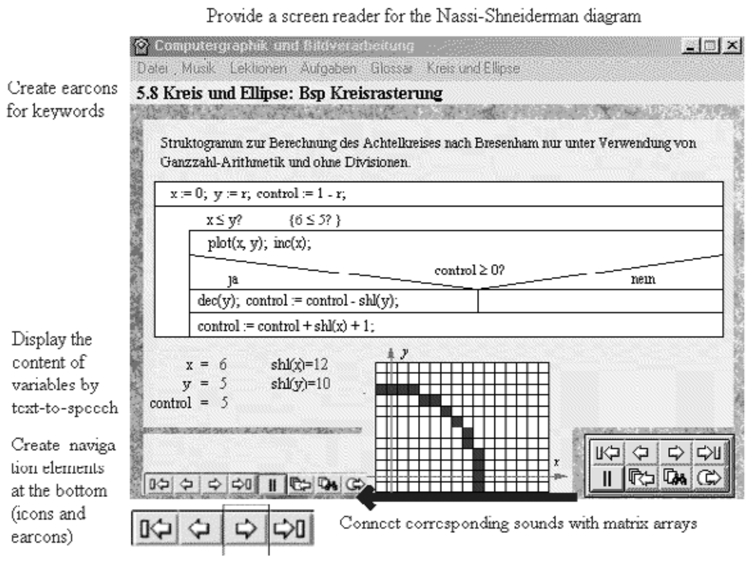
Korrekt!

Fig. 3. Visualized aural and graphic representation of a raster curve

Now we will give further examples of how to animate standard graphic algorithms in two dimensional pixel geometry using different perception channels. There are basically two different ways to proceed. The common way is to show how the algorithm works using its logical representation, e.g., the Nassi-Shneiderman diagram or other control flow charts. Once, a pseudo code of the algorithm is given, a small window can be moved over the diagram lightening the current command line being executed. Thus, the control flow is animated and assignments, branches and loops are highlighted.

A completely different method is based on displaying the manipulated data. However, several points must be mentioned in regards to this method. First, an adapted visual representation of the data must be developed. Loop variables, flags, stacks and registers are interesting candidates for display. It is necessary to scale data in order to obtain coarser structures, to slow down the working algorithm, and to find an adequate representation of the output. We postulate an interactive user control allowing the user to step forward and backward within the algorithm, to redo steps and to modify the step width. This can be done with the concept keyboard.

Our second example is the Bresenham algorithm which digitizes lines, circles, and ellipses (see Figure 4). At the same time, the growing circle or ellipse altering the contents of typical variables can be seen. First, the growing curve is translated into appropriate intuitive sounds accompanied by spoken text concerning the contents of the variables. Another view presents the Nassi-Shneiderman diagram via text-to-speech. Finally, it is necessary to group the control elements at the bottom of the screen to simplify the navigation.


Provide a screen reader for the Nassi-Shneiderman diagram
Create earcons for keywords
Display the content of variables by text-to-speech
Create naviga tion elements at the bottom (icons and earcons)
Connect corresponding sounds with matrix arrays
Computergraphik und Bildverarbeitung
Datei Musik Lektionen Aufgaben Glossar Kreis und Ellipse
5.8 Kreis und Ellipse: Bsp Kreisrasterung
ja
nein
x
y

Struktogramm zur Berechnung des Achtelkreises nach Bresenham nur unter Verwendung von Ganzzahl-Arithmetik und ohne Divisionen.

```
x := 0; y := r; control := 1 - r;
x ≤ y?        {6 ≤ 5? }
    plot(x, y); inc(x);
    control ≥ 0?
    ja: dec(y); control := control - shl(y);
    nein:
    control := control + shl(x) + 1;
```

```
      x  =  6      shl(x)=12
      y  =  5      shl(y)=10
control  =  5
```

Fig. 4. Enhanced representation and interaction for the Bresenham algorithm

ViACoBi is complemented by another teaching system which presents comparable solutions for spatial data structures like bintrees, quadtrees or octrees; this system will be provided with a haptic interface in the future. These new features will be implemented in a new version of ViACoBi.

4.2 Sorting Algorithms

Usually algorithm animation maps elements on rectangles with heights proportional to sizes. Alternatively, a sorted array of numbers for a sorting algorithm can be represented by a sequence of sounds with ascending frequencies. For an example like Heapsort it is conceivable to replace the node contents with sounds and then explore the data structure using a concept keyboard, a pen or a joystick together with a tracking function and attracting nodes. The program code can be augmented by auditory messages concerning the current running procedure and its current parameter values, the values of control variables within the loops, the result of comparisons in conditions, the result of changing memory contents, and the progress in work. The last point is a very important one because the preceding ones only concern local actions without contributing to a generalized view on the algorithm or data structure. These messages can be generated automatically like some debugging information and can be displayed graphically or by text-to-speech. So our focus changes from algorithm visualization to algorithm explanation.

4.3 Graph Structures

In a graph each vertex can be complemented by an explanation concerning the content, the parent and child nodes and the global position within the graph. Outgoing and incoming edges are mentioned. There are many classical graph algorithms to construct shortest paths or spanning trees. When we want to go from point A to point B, a standard visualization tool proposes a two dimensional road map containing start and end point but also many further details. So the user must collect the relevant information, discard all roads which do not serve, and extract an adequate solution path. A better way is to preprocess the map and to use a spanning tree or only good paths near A and B. However, the simplest way is to announce through an acoustic channel a sequence of local landmarks and directions to reach point B.

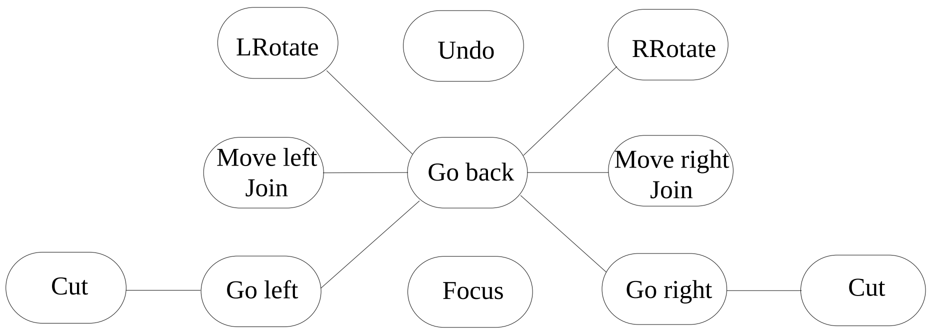


Fig. 5. Concept keyboard for AVL-trees

Balancing an AVL-tree after the insertion or deletion of vertices is a problem which is addressed by several tools for algorithm animation. For instance, by using standard right or left rotations the tree can be rebalanced. To do this work and to navigate on the tree we propose the concept keyboard given in Figure 5.

4.4 Didactic Network

In their paper [2] N. Baloian et al. discuss the concept of didactic networks. In such a network, the teacher collects multimedia material which is organized as the vertices of a directed graph and presented to the students, while the teacher follows a certain instruction path. The network provides the following types of nodes: Graphic, Animation, Audio, Video, Text, Discussion, Individual and Group Work. Nodes are linked by different typed edges with labels corresponding to standard teaching methods, such as 'introduces to', 'refined by', 'explained by', 'exemplified by' and 'summarized by'.

The lesson manager controls the visualization and presentation tools. Relevant parts of the network are constantly displayed to support the user. Different teaching methods lead to different weights of the edges. A spanning tree is calculated to overcome the very complex graph structure. The teacher steps through the tree visiting the most important nodes and presenting their contents. It is an interesting idea to develop appropriate concept keyboards for navigating within the network to explore the logical structures which brought to the teacher's mind by different perception channels. The concept keyboard supports the teacher's navigation through the learning material in a didactically coherent way according to a predefined teaching or learning strategy (see Baloian, Hoppe, Luther [3]).

4.5 Design Principles

Our considerations can be summarized in the following list of design principles which can be used to add other transmission/reception modes in parallel to classical visualization:

– Enable users to explore objects represented by digital shapes or three dimensional octree-models and neighborhoods by pointing to or grasping them in order to generate images in the mind.
– Transform graphs into acoustic structures; supply nodes with text, images, pictures, and links with captions which can be presented by text-to-speech.
– Use earcons and icons in parallel; use frequencies to represent coordinate values and different instruments for different axes.
– Provide for a graphic tablet and pen or special concept keyboards to traverse a graph structure.
– Develop for each application a suitable concept keyboard by redefining keys, creating special button schemes on a graphic tablet or introducing a new device custom-built for the application.
– Use commercial or free screen readers. They help visually impaired people navigate within information systems.
– For algorithm animation divide the screen into two parts: Place on the right side comments on the algorithm or a formal description which can be presented by text-to-speech and on the left side a visualization enhanced with acoustic motifs (directions or dimensions can be represented by different instruments and positions by tonal pitches). Thus, a natural slow down is obtained by spoken explanations.
– Introduce modern haptic interfaces like gloves or wireless ultrasonic joysticks. They provide a new feeling of three dimensional geometric data structures as was reported in [14, 21].

However, all these concepts lead to a stepwise or local processing of the model. Therefore, an important question must be answered: Is it possible to achieve a global mental image of a static or dynamic scene only from local exploration? Furthermore, we must develop evaluation strategies and best practice examples to prove that augmented interfaces and complementary perception channels provide better comprehension.

5 Ideas for Automatic Generation of the Concept Keyboard

A concept keyboard allows users to traverse a data structure and/or perform certain procedures or functions on it step by step. Ideally, the concept keyboard would be generated directly from the code implementing the data structures and algorithms over them. However, then we would too often be confronted with declarations of data structures defining fields and operations needed for implementing the whole system but making the understanding of important ideas and issues more difficult. The use of parameters in the implementation of methods for traversing or modifying the data structure may also complicate the generation of a simple but meaningful concept keyboard. To overcome these problems, we propose using an interface file which defines the data structures and procedures in a suitable way, making the generation of a keyboard easier and "cleaner". Of course, how "clean" and meaningful (that is, how helpful for understanding the data structure and their algorithms) the resulting keyboard will be depends on how the interface file is written, but this seems to us to be unavoidable. The definition of the interface file should be made according to the following rules:

- Define the data structures containing only the relevant information which is needed to understand the general problem and its solution.
- Define a number of methods or operations having only the current node (or array element) as implicit parameter (like "this" in a Java class definition). Other nodes should not appear as parameters. Instead of this, more operations should be defined. (For example, instead of rotate(x) with x being the left or the right child, a rotateLeft() and a rotateRight() operation should be declared.)
- Associate with each operation the corresponding call to the method implementing it.

This last condition may cause the programmer to change the original code of the implementation but this should not demand too much work if the code has been reasonably written. The interface file definition language should be XML (eXtensible Markup Language), which is a meta-syntax used to declare Document Type Definitions (DTDs) for existing and new computer markup languages. The focused DTDs are intended for User Interface-oriented structural, textual, graphic, acoustic, or tactile renderings. XML promotes the creation of accessible documents which can be equally well understood by their target audience regardless of the standard or concept keyboard device used to access them. XML permits the simple yet flexible definition of structured documents. This feature makes the description of data structures and functions with this language very easy. This, added to the fact that XML is a standard, makes it the right candidate to be used as the interface definition language for concept keyboards.

6 Related Tools for Blind Users

A number of systems have been developed with the aim of being used by people with disabilities. While most systems targeted for the hearing impaired are oriented to train people by developing the necessary skills to overcome their disabilities, a considerable proportion of the systems for blind people aim to increase greatly the accessibility to current computing resources which are based on graphic user interfaces, such as games and web navigators.

For blind people screen reader software allows access to graphic user interfaces by providing navigation as well as a Braille display or speech synthesized reading of controls, text, and icons. The blind user typically uses the tab and arrow controls to move through menus, buttons, icons, text areas, and other parts of the graphic interface. As the input focus moves, the screen reader provides Braille, speech, or non-speech audio feedback to indicate the user's position. Blind users rarely use a pointing device, and as discussed above, typically depend on keyboard navigation. A problem of concern for blind users is the growing use of graphics and windowing systems [5].

The browser BrookesTalk reads out the Web page, using speech synthesis in word, sentence, and paragraph mode, and offers different views of the page to simulate 'scanning' [25]. Different views of the page take the form of keywords, summary or abstract and are derived using information retrieval and natural language processing techniques. BrookesTalk also offers a special search facility, a configurable text window for visually impaired users and a standard visual browser so that blind users can work alongside other people who can fully utilize a standard graphic interface.

The HOMER UIMS approach by Anthony Savidis and Constantine Stephanidis [22, 23] develops dual user interfaces for the integration of blind and sighted. HOMER supports the integration of visual and non-visual interaction objects and their relationships. In this context, a simple toolkit has been also implemented for building non-visual user interfaces and has been incorporated in the HOMER system. A Dual User Interface is characterized by the following properties: It is concurrently accessible by blind and sighted users and the visual and non-visual metaphors of interaction meet the specific needs of sighted and blind users, respectively. At any point in time, the same internal (semantic) functionality is made accessible to both user groups through different interactive channels.

Education for blind children uses special devices like a touch pad, speech synthesis or Braille displays. Another idea is to use concept keyboards which implement different actions to interact with applications. There are excellent tools for translating text to Braille; however, graphics production has problems creating tactile graphics on screen or printing it on special embossed or swell paper. A specialized thermal printer produces output on swell paper. Many modern Braille printers have a graphics mode where dots are embossed in a regular raster. It is even possible to leave the raster in order to create "smooth" curves. Other solutions print graphics directly from a program and handle the conversion to a relief image. We prefer a combination of touch, text-to-speech and sound landscapes, an access chosen in the European project TACIS (Tactile Acoustic Computer Interaction System). Together with Windows-based screen readers, Braille displays allow the visually impaired person to obtain a tactile layout of the desktop, applications, and windows [13].

AudioDoom [14, 21] allows blind children to explore and to interact with virtual worlds by using only spatialized sound. It was inspired by traditional Doom games where the player has to move inside a maze discovering the environment and solving problems posed by objects and entities that inhabit the virtual world. In doing so, it emphasizes the sound navigation throughout virtual spaces in order to develop cognitive tasks to enhance spatial orientation skills in blind children. AudioDoom is manipulated by using a wireless ultrasonic joystick or a labeled keyboard.

In the same context the sonic concentration game proposed by Roth et al. [20] contains several different levels offering pairs of geometrical shapes to be matched. To represent geometrical shapes it is necessary to build a two-dimensional sound space. Each dimension corresponds to a musical instrument and Raster points correspond to pairs of frequencies on a scale. Moving horizontally from left to right is equivalent to a frequency variation of the first instrument, and moving vertically to a frequency variation of the second one.

Another mode presents a shape by moving sound in a special plane. The third dimension can be represented by means of frequency. The Doppler effect can be used to enhance front and back differences.

Haptics is a technology that provides sensing and control through touch and gesture. It presents interface equipment for accessing and manipulating data normally available only through visual representations for seeing people. A haptic system must sense and analyze the forces applied by the user and deliver a physical sensation back to the person in real time. This kind of systems allows users to explore all sides of a virtual 3D object, to move and to turn the object freely in three orthogonal space coordinates and around three rotational axes, and to track force and to provide force-feedback sensations in multiple dimensions.

A haptic interface is a hand-held computer-controlled motorized device displaying information to human tactile and kinesthetic senses. It works bidirectionally, accepting input from the user and displaying output from the computer. Including haptics in our scenario offers a further important channel, parallel to visual and aural communication, which can substitute for the other sensorial streams, in particular graphic interfaces. For example, by producing forces on the user's hand depending on both the user's motions and properties of the icons under the cursor, touchable representations of the screen objects can be created [19]. Gloves with embedded sensors provide hand-grasping actions or whole-hand sensitivity. Realistic sounds of haptic interaction can be easily synthesized, enabling systems to convey many haptic perceptions, such as hardness, material, texture, and shape.

7 Conclusion

We have presented new concepts for enhancing standard visual interfaces with aural or haptic components to convey logical structures or data types to human minds. Optional dual or multiple interfaces enhance the human-machine interaction and support sensory-disabled people. It is important to separate control elements and visual objects by mean of an adequate concept keyboard which will be used to process data structures and geometric models. To achieve meaningful visualization, we focus on important information after a suitable abstraction process. The design guidelines given will be integrated into new versions of ViACoBi and a cooperative AudioDoom for the target groups mentioned above.

Acknowledgements. This research and common multi-media software development for disabled (AudioDoom for blind people and Whisper to enhance proficiency in speaking for postlingually deaf people) are being carried out by the authors in a current German and Chilean project funded by the German ministry BMBF and the Universities of Duisburg and (Santiago de) Chile [4, 14, 21, 24]. This cooperation gives us the opportunity to prove or disprove the notion that concept keyboards are useful for blind users. We thank Prof. Sánchez for valuable suggestions concerning the AudioDoom system and our collaborators Dr. W. Otten and C. Wans for their persistent valuable support during the development and evaluation of Whisper, discussing these ideas and making the material of their research available to us.

We would like to express our gratitude to our reviewers, who help us to reorganize the paper and to clarify our ideas and concepts.

References

1. Archambault, D., Burger, D.: TIM (Tactile Interactive Multimedia): Development and adaptation of computer games for young blind children. Workshop on Interactive Learning Environments for Children, Athens, Greece, March 1-3 (2000) http://ui4all.ics.forth.gr/i3SD2000/Archambault.PDF
2. Baloian, N., Pino, J., Hoppe, H. U.: Intelligent navigation support for lecturing in an electronic classroom. In Lajoie, S. und M. Vivet (Hrsg.). *Artificial Intelligence in Education – Open Learning Environments: New Computational Technologies to Support Learning, Exploration, and Collaboration.* Amsterdam et al.: IOS/Omsha (1999) 606-610
3. Baloian, N., Hoppe, H.U., Luther, W.: Structuring Lesson Material to Flexibly Teaching in a Computer Integrated Classroom. GI-Workshop der Fachgruppe "Intelligente Lehr-/Lernsysteme", Dortmund, Germany, October 8-11 (2001). Research Report 763, University of Dortmund, ISSN 0933-6192, 187-194
4. Baloian, N., Luther, W., Sánchez, J.: Modeling Educational Software for People with Disabilities: Theory and Practice. Submitted to ASSETS 2002 conference
5. Bergman, E., Johnson, E.: Towards Accessible Human-Computer Interaction, in "Advances in Human-Computer Interaction", Volume 5, Jakob Nielsen, Editor (1995)
6. Bissell, R. E.: Music and Perceptual Cognition[1]: Journal of Ayn Rand Studies, Vol. 1, No. 1, Sept. 1999. http://www.dailyobjectivist.com/AC/musicperceptualcognition6.asp
7. Blattner, M., Sumikawa, D., and Greenberg, R.: Earcons and icons : Their structure and common design principles. Human-Computer Interaction 4, (1) (1989) 11-44
8. Brewster, S.A.: Using earcons to improve the usability of a graphics package. HCI'98, People and Computers XIII, Sept. 1-4 (1998) Sheffield Hallam University, Sheffield, UK

9. Diehl, St., Görg, C., and Kerren, A.: Preserving the Mental Map using Foresighted Layout. Proc. Eurographics – IEEE TCVG Symposium on Visualization, VisSym, 28-20 May 2001, Ascona, Switzerland
10. Janser, A.: Ein interaktives Lehr-/Lernsystem für Algorithmen der Computergraphik. In: Schubert, Sigrid (Hrsg.): Innovative Konzepte für die Ausbildung, Informatik aktuell, Springer, Berlin (1995) 269-278
11. Janser, A.: Entwurf, Implementierung und Evaluierung des interaktiven Lehr- und Lernsystems VIACOBI für die Visualisierung von Algorithmen der Computergraphik und Bildverarbeitung. Logos, Berlin (1998) ISBN 3-89722-065-2
12. Kramer, G. (ed.): Auditory Display - Sonification, Audification, and Auditory Interfaces. Addison-Wesley (1994)
13. Lange, Max O.: Tactile Graphics - as easy as that. CSUN's 1999 Conference Proceedings. March 15-20, Los Angeles, USA. http://dinf.org/csun_99/
14. Lumbreras, M., Sánchez, J.: Interactive 3D Sound Hyperstories for Blind Children. *CHI '99*, Pittsburg PA, USA (1999) 318-325
15. Luther, W.: Algorithmus-Animation in Lehr- und Lernsystemen der Computergraphik. Diehl, St., Kerren, A. (Hrsg.). GI-Workshop SV2000. Dagstuhl, 11.-12.5.2000, 103-114
16. Mereu, S., Kazman, R.: Audio enhanced 3D interfaces for visually impaired users. Proc. ACM CHI 96 (1996) 72-78
17. Morley, S., Petrie, H., O'Neill A.-M., McNally, P.: The Use of Non-Speech Sounds in a Hypermedia Interface for Blind Users, in Edwards, A.D.N., Arato, A., and Zagler, W.L. (Eds.): 'Computers and Assistive Technology'. Proc. ICCHP'98. XV. IFIP World Computer Congress (1998) 205-214
18. Norseen, J.: Images of Mind: The Semiotic Alphabet (1996) http://www.acsa2000.net/john2.html
19. Ressler, S., Antonishek, B.: Integrating Active Tangible Devices with a Synthetic Environment for Collaborative Engineering. Proc. 2001 Web3D Symposium. Paderborn, Germany, Febr. 19-22 (2001) 93-100
20. Roth, P., Petrucci, L., Assimacopoulos, A., Pun, Th.: Concentration Game, an Audio Adaptation for the blind. CSUN 2000 Conference Proceedings, 20-25 March, Los Angeles, USA (2000) http://www.csun.edu/cod/conf2000/proceedings/gensess_proceedings.html .
21. Sánchez, J., Lumbreras, M: Usability and Cognitive Impact of the Interaction with 3D Virtual Interactive Acoustic Environments by Blind Children. 3rd Int. Conf. on Disability, VR and Assoc. Technologies Alghero, Sardinia, Italy, 23-25 Sep. 2000
22. Savidis, A., Stephanidis, C.: Developing Dual User Interfaces for Integrating Blind and Sighted Users : HOMER UIMS, Chi'95 conf. proceedings, Denver, CO (1995) 106-113
23. Savidis, A., Stephanidis, C., Korte, A., Crispie, K., Fellbaum, K.: A generic direct-manipulation 3D-auditory environment for hierarchical navigation in non-visual interaction. Proc. ACM ASSETS 96 (1996) 117-123
24. Wans, Cl.: Computer-supported hearing exercises and speech training for hearing impaired and postlingually deaf. Assistive Technology Res. Series, 6 (1). IOS Press (1999) 564-568
25. Zajicek M., Powell C., Reeves C.: A Web Navigation Tool for the Blind, ASSETS'98, 3rd ACM/SIGGRAPH Conf. on Assistive Technologies, Los Angeles, USA (1998) 204-206

The *rube* Framework for Personalized 3-D Software Visualization

John F. Hopkins and Paul A. Fishwick

Department of Computer and Information Science
University of Florida, Gainesville, FL, USA

Abstract.

In this chapter, we discuss a software modeling and visualization framework called *rube*[†]. This framework facilitates the creation of three-dimensional (3-D) software visualizations that integrate both static software architecture and dynamic real-time operation. A unique aspect of *rube* is that it does not tie developers down to a set of predefined symbols, objects, or metaphors in their visualizations. Consequently, users have the freedom to develop their own representations. The *rube* framework's general approach to software modeling and representation are discussed. Next, a simple example is developed according to *rube's* systematic modeling and visualization process. Lastly, benefits of the framework and future directions are discussed.

1 Background

Modeling plays an important role in many computing tasks, including software engineering and software visualization (SV). The first two phases of the modeling process involve system understanding and model representation. In the current context, a *system* is any real-world (e.g., ecosystem) or abstract (e.g., database) entity, and a *model* represents the discrete objects and interactions between objects in the system. If readers will indulge us, we consider the terms *software*, *program*, and *model* to be more or less conceptually equivalent unless otherwise noted for the purpose of discourse in this chapter. Likewise, we consider the terms *software developer*, *programmer*, and *modeler* to be more or less conceptually equivalent unless otherwise noted.

Modelers come to understand, predict, and analyze a system based on the models that they construct for it. The model represents a key medium that links modelers to the phenomena. Thus, the model's representation plays an important role as an interface to its users. Although historically there has been a rich variety of textual and diagrammatic approaches to model representation, there has been little systematic accommodation of personal preference in these approaches.

Our culture is driven in part by economy of labor and materials, and personalization is held back primarily for these economic reasons. However, today's

[†] *rube* is a trademark of Paul A. Fishwick and the University of Florida.

economy is at a stage where personalization has become more feasible and some trends in personalization are developing in both media and human-computer interfaces. For our immediate purposes, we do not draw a distinction between customization and personalization, treating both as facets of aesthetic choice. An example of personalization in human-computer interfaces is the current proliferation of customized *skinz* in window-based graphical user interfaces (GUIs). The renewed focus on accommodating the individual in media and user interfaces suggests a corresponding accommodation of personal preference in model representation and by extension, 3-D SV.

There may be some advantage to be gained from personalization in 3-D visualization. To illustrate, a frequent occurrence during the development process is that one or more abstract data types or functions are created. If a developer finds value in 3-D visualization and would like to visualize an abstract function such as a sorter, he or she may arbitrarily decide to visualize it as a green pyramid. Instead, it may be possible to create a more elaborate visualization for the sorter. For example, the developer might be able to visualize the sorter as an animated person who is sorting boxes. If the developer uses the animated sorting person, he or she has made an analogy between the abstract sorter and the concrete, real-world person. If the sorter is visualized in this fashion, there is no need to memorize the previous mapping of the green pyramid to the sorter. The visualization of the person and the analogy introduced now provide this mapping implicitly. In effect, the visualization provides a semantic cue as to the object's function. This sort of visualization, then, may serve as a form of implicit documentation. It would be difficult to support the argument that the green pyramid visualization is preferable to the animated sorting person on grounds other than the additional effort it would take to produce the animated person. Finally, the extra effort required to produce a personalized 3-D visualization should decrease to a minimal level with time and advances in technology, so that a cost/benefit analysis should eventually become favorable.

There is some empirical evidence that shows the value of self-construction in visualization. For example, the use of metaphor in diagrams has been shown to provide some mnemonic assistance, which appears to be greatest when the user of the diagram constructs his or her own metaphor [1]. In addition, it has been empirically established that the process of actively constructing one's own visual representations is more beneficial than passively viewing someone else's visual representations [2].

Fishwick [3, 4] has been developing a modeling framework called *rube* in which users develop both static and dynamic 3-D model visualizations in parallel with other modeling efforts. What sets *rube* apart from similar work is that these visualizations can be highly customized by the user. This chapter discusses *rube's* approach to modeling and representation. To illustrate the *rube* modeling process, a systematic example of model development is presented. The example is a simple Finite State Machine. Finally, the benefits of the approach and future directions are discussed.

2 The *rube* Framework

2.1 *rube* Framework's Precursor: Object-Oriented Physical Multimodeling

In previous research, Cubert et al. [5], Fishwick [6], and Lee and Fishwick [7] have worked on the development and implementation of an object-oriented simulation application framework. *Object-Oriented Physical Multimodeling* (OOPM) is a system that is a milestone product of this previous research [5, 7]. OOPM extends object-oriented program design through visualization and a definition of system modeling that clarifies and strengthens the relationship of model to program [3]. The "physical" aspect of OOPM reflects a model design philosophy that recommends that models, components, and objects should be patterned after the structure and attributes of corporeal objects.

Within OOPM, programs are multimodels [6, 8, 9, 10]. A multimodel is defined as a hierarchically connected set of dynamic behavioral models, where each model is of a specific type and the set of models may be homogeneous or heterogeneous [5, 8]. The basic dynamic behavioral model types are numerous and include Conceptual Model (CM), Finite State Machine (FSM), Functional Block Model (FBM), System Dynamics Model (SDM), Equation Constraint Model (ECM), Petri Net (PNET), Queuing Net (QNET), and others [8]. OOPM supports the creation and execution of several of these model types including CM, FSM, FBM, SDM, ECM, and RBM. The dynamic behavioral model types are freely combined in OOPM through the process of multimodeling, which "glues together" models of same or different type [5].

An example of a multimodel based on a real-world system might be the following: Assume that a group of people is standing in a straight line in front of a single ticket booth. The line is a simple queuing network (QNET), with a queue (the line), queued entities (the people), and a server (the ticket booth). Now, assume that we would like to describe the state of each person waiting in the line as "stopped, moving, or being served." To model these states, we could incorporate a finite state machine (FSM) within each person. There would be three states in each FSM: *stopped*, *moving*, and *being served*. Events that are happening in the queuing network would trigger transitions between states in each person's FSM. If the line is moving, the FSMs for people in the line transition into the *moving* state. When the line stops, the FSMs transition into the *stopped* state. If a person is the next in line for waiting for service, that person's FSM will transition from *stopped*, to *moving*, to *being served*. This completes the example multimodel, which demonstrated a hierarchical arrangement of FSMs within entities that were part of a QNET.

The OOPM system has some other noteworthy features. One feature is its 2-D GUI, which facilitates model design, controls model execution, and provides 2-D output visualization [5]. Another feature is a model repository that facilitates collaborative and distributed model definitions, and that manages object persistence [5].

2.2 The Goals of *rube*

The *rube* framework and OOPM share many characteristics. For example, they both make use of the previously listed dynamic behavioral model types within a multimodeling framework. However, *rube* research and development (R&D) moves OOPM concepts into the third dimension and expands on them. Specifically, the goals of *rube* R&D are:

1. To create a model design methodology and a software system that supports a *separation* of dynamic model specification from presentation and visualization.
2. To work with the Fine Arts community (e.g., university Digital Arts and Sciences programs) in creating more personalized and aesthetic presentations. The *rube* framework supports this effort by promoting the integration of modeling with developer-defined visual and audible elements.
3. To enable specification of dynamic models for use in a wide variety of systems needs, one of which is programming (and others are models used for simulation). One physical manifestation of this goal is a publicly available World-Wide-Web (WWW) based toolkit composed of reusable, generic, 3-D model components based on the basic dynamic behavioral model types along with a model repository composed of fully developed models.

2.3 The *rube* Development Environment

The *rube* development environment is implemented in XML (eXtensible Markup Language), and includes a 3-D, web-based GUI [3] that controls a Model Fusion Engine [11]. The Model Fusion Engine supports the fusion of geometry models, dynamic models (e.g., FSM, FBM, and others as previously listed), and their scripted behaviors [11]. The fusion process merges a geometric *scene file* and a *model file* [11]. The *scene file* contains a user-defined VRML (Virtual Reality Modeling Language) world either created in a 2-D text editor or exported from other 3-D software such as *CosmoWorlds* or *3D Studio Max* [11]. The *model file* is a user-defined XML file that defines connectivity between objects and the behavior of the dynamic model types [11]. Each dynamic model is modularized and used as a separate library [11]. When the Model Fusion Engine finishes merging the scene and model files, it generates an X3D (eXtensible 3D) file [11]. This file is then translated into a VRML file that can be displayed in a VRML browser such as such as *Blaxxun Contact*, *Parallel Graphics' Cortona*, and *CosmoPlayer* [11].

The GUI is shown in Fig. 1. In the lower part of the window, a user can specify or upload user-defined *scene* and *model* files [11]. In the upper part of the window, the newly created 3-D dynamic model is displayed with a VRML browser [11]. It is important for readers to note that *rube* does not implement a 3-D "programming" GUI that, for instance, allows a user to construct a network of 3-D objects that would be parsed by the GUI to automatically generate an executable program, such as Najork's *CUBE* [12]. In addition, unlike *CUBE*, *rube* does not possess a formal "3-D syntax," nor is it a set of 3-D representations of primitive data types and atomic operations.

One major distinguishing feature of *rube's* modeling architecture is that it separates geometry from inter-object semantic relations [11]. Any *scene file*, which represents geometry and appearance, can be used along with any *model file*, which contains information about relations and behaviors of the model [11].

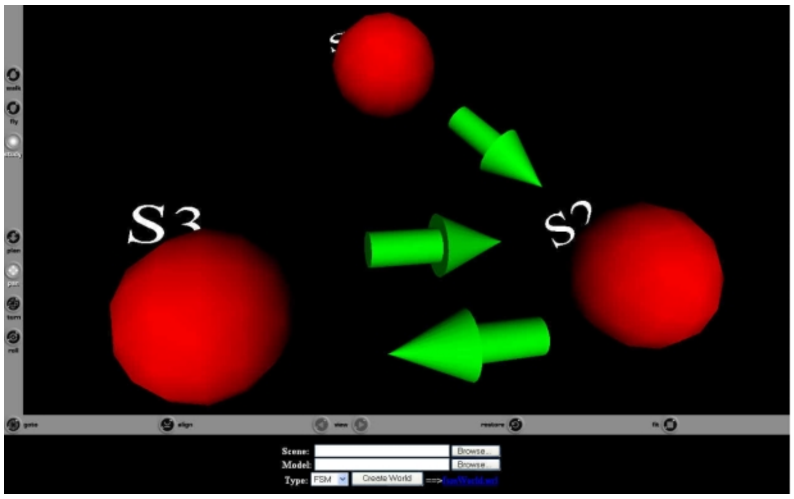


Fig. 1. *rube's* GUI.

The *rube* development environment allows users to either create or reuse existing 3-D objects for the *scene file*, and allows users to create or reuse dynamic models for the *model file* [11]. The freedom of defining and creating 3-D objects has been given completely to the model author [11]. Objects can be personalized and made culturally or aesthetically meaningful [4, 11, 13].

2.4 Placing *rube* in a Frame of Reference with Respect to Software Visualization

To give readers a better idea of the characteristics of the *rube* framework, we classify it according to the program visualization system taxonomy given by Roman and Cox [14]. Within this taxonomy, there are three possible roles to be played: *programmer*, *animator*, and *viewer* [14]. A *rube* developer is both the *programmer* and *animator*, and anyone may be a *viewer*. At this time, we consider the primary viewing audience to be either the model author or someone who is familiar with modeling in the context of *rube*. This slant may change in the future, and we have been investigating both novice modelers and artistic communities as possible users/audiences for the *rube* framework and its models. Continuing with the axes of the taxonomy [14]:

- *Scope*: *rube* does not automatically transform model code into page layouts, such as flowcharts or statement-level diagrams. It is capable of showing model data and control states, but the implementation of these capabilities is up to the programmer/animator. Since *rube* is primarily an event-based system, it is best padapted by programmers/animators for relating model behavior.
- *Abstraction*: *rube* is capable of direct, structural, and synthesized model representation. However, *rube* primarily encourages structural representation. Again, the implementation of these capabilities is up to the programmer/animator.

It is possible to implement "zooming" capabilities to provide low and high levels of abstraction to a user that is navigating throughout a world.

- *Animation Specification Method*: *rube* relies heavily on annotation by the programmer/animator. Predefinition, declaration, and manipulation do no play a role in *rube*.
- *Interface*: *rube* inherits its graphical capabilities from VRML and its successor, X3D. Thus, it can specify simple objects, compound objects, visual events, and worlds. Within worlds, both absolute and constraint-based positioning are permitted. Multiple worlds (i.e., separate windows with separate objects that all represent alternate views of the same model) are possible, but are not a not a focus of *rube*. Interaction with the world by the viewer is managed through the VRML browser and is programmer/animator defined. Interactive capabilities are often implemented in the form of predefined controls. These controls can be embedded in the image.
- *Presentation*: *rube* is extremely flexible concerning interpretation of graphics. Any accompanying text-based explanations must take into account the intended audience. Since the audiences for *rube* models are not yet well defined, we leave this issue to programmers/animators. Programmers/animators need to be aware that others may not easily grasp their display customizations, perhaps performed without regard to related conventional or "obvious" styles of presentation. Accompanying detailed text-based explanations may be necessary. A *rube* visualization is capable of showing explanatory events and orchestrations. The incorporation of aesthetic visual and audible elements in models are encouraged by *rube*.

Najork and Brown's *Obliq-3D* is a high-level, fast-turnaround system for building 3-D animations that consists of an interpreted language that is embedded into a 3-D animation library [15]. There are some similarities between *rube* and *Obliq-3D*. The languages used in both systems for specifying graphics objects have similar structure, expressive power, animation, and interactive capabilities. Both systems use interpreted languages (for *rube*: XML, VRML, X3D and etc.), so both are "fast turnaround." There are also some significant differences between *rube* and *Obliq-3D*. The *rube* development system is web-based and portable (e.g., HTML, XML, VRML, X3D), while *Obliq-3D* is not (e.g., X-Windows, Microsoft Windows, Modula-3, Obliq). VRML, X3D, and XML are not strictly OO languages, while *Obliq* is. It should be easier to create graphical structures, especially compound structures, in a free-form environment in *rube* using third party tools that export VRML. In addition, geometric structures can be easily reused for any purpose in *rube*. Sound can be incorporated as part of *rube* models, while *Obliq-3D* does not mention this capability. Finally, *rube* and *Obliq-3D* were designed around somewhat divergent goals: *rube* is more focused on aesthetics and modeling in a formal sense.

2.5 The Steps of the *rube* Modeling Methodology

The *rube* modeling and visualization methodology proposed by Fishwick [4] consists of the following five steps:

1. *Choose system to be modeled*: This could be anything from a system in the real world (e.g., the Everglades ecosystem), to a typical software system (e.g., database).
2. *Select structural and dynamic behavioral model types*: Here, modelers specify the dynamic behavioral model types to be used in designing the multimodel. These include CM, FSM, FBM, SDM, and others as previously listed. Next, modelers specify the dynamics and interactions between the different models.
3. *Choose a metaphor and/or an aesthetic style*: Here, modelers develop their own custom metaphors for the phenomena that they are modeling. It is preferable that these metaphors have some readily apparent relationship to the phenomena being modeled, but the presence of such a relationship is not required by *rube*. For example, modelers may choose an architectural metaphor. Within architecture are many different aesthetic styles to choose from like Romanesque, Baroque, and Art Deco.
4. *Define mappings/Develop analogies*: In this step, the modeler develops a careful and complete mapping between the structural and dynamic behavioral model type components and the metaphorical and stylistic components. Although the *rube* development environment itself does not specifically support the "automatic" or "assisted" mapping of a software system to a visualization, it does offer guidelines for some common modeling and programming constructs [4].
5. *Create model*: Here, the modeler combines the models and mappings generated in the previous to synthesize the multimodel.

These original steps were derived before the current work in XML. Our current work assists the users in these steps as follows. For step 2, there are a set number of dynamic mode types planned for *rube* and the formal XML schema specification for two of them (FSM and FBM) are underway. Step 3 currently remains manual. For step 4, there are guidelines but no programmatic assistance. Step 5 includes significant assistance in the form of the Model Fusion Engine.

3 *rube* Example World

The following world briefly addresses the high-level development of a simple model and its visualization in *rube*.

Step 1: Choose System to be Modeled
We will specify a simple light bulb system that can be in three different states. First, we must connect the bulb, by way of a ceramic base, to the wall socket. This state is represented by an initial state of "disconnected." Once the light is connected, it moves to a second state of "off." From there, it moves to "on" if a chain is pulled. If the chain is pulled again, the light goes "off," and so on.

Step 2: Select Structural and Dynamic Behavioral Model Types
The system that we chose in the previous step can be modeled well with an FSM. Before we begin the development of our FSM, let us first give a basic formal definition. A FSM is described by the set $\langle T, U, Y, Q, \Omega, \delta, \lambda\rangle$, where

- T is the *time base*. $T = \Re$ (real numbers) for continuous time systems and $T = Z$ (integers) for discrete time systems.
- U is the *input set* that contains all possible values that can be used as input to the system.
- Y is the *output set* that contains all possible values that can be output by the system.
- Q is the countable *state set*.
- Ω is the set of *acceptable input functions*.
- δ is the *transition function,* $\delta: Q \times \Omega \rightarrow Q$.
- λ is the *output function,* $\lambda: Q \rightarrow Y$.

A simple FSM that describes our light bulb system is shown in Fig. 2. S1 represents "disconnected," S2 represents "off," and S3 represents "on." Connecting the bulb to the ceramic base activates the S1 $\rightarrow$ S2 transition (labeled with a "1"). Pulling the chain to turn the light "on" and "off" alternately activates the S2 $\rightarrow$ S3 and S3 $\rightarrow$ S2 transitions (both labeled with a "2"). Our FSM is defined as follows:

- FSM = $\langle T, U, Y, Q, \Omega, \delta, \lambda\rangle$
- $T = Z_0+$
- $U = \{ 1, 2 \}$
- $Y = Q$
- $Q = \{$ S1 (start state), S2, S3 $\}$
- $\Omega = 1$ for t_0, and 2 for all other T
- $\delta: Q \; x \; \Omega \rightarrow Q$
- $\lambda: Q \rightarrow Y$

In the first time step, the FSM will change state from S1 to S2. Thereafter, on each time step, the FSM's state will alternate between S2 and S3.

Step 3: Choose a Metaphor and/or an Aesthetic Style
Here, it is likely that a user would choose a single metaphor to represent the system, or perhaps a group of metaphors and sub-metaphors to represent a complex system with many components. In this process, there would be no more than one metaphor to map to each major model component in the following step (i.e., step 4). However, for the sake of discussion, we choose two metaphors that will map to our single FSM and we will show the application of these two metaphors to our example FSM in parallel during the remainder of the modeling steps. This approach will show the flexibility of *rube*. One metaphor will involve water tanks, pipes, and water, and the other will involve gazebos, walkways, and a person. Both metaphors are conceptualized in 3-D.

Step 4: Define Mappings/Develop Analogies
The mappings between the water tank/pipe metaphor and the FSM are simple:

- Water tanks represent states in our FSM. When a water tank is full, the FSM is in the state represented by that water tank. Only one tank may be full at a time. Since our FSM has three states, we will need three water tanks. Each water tank will correspond to a specific state in our FSM.

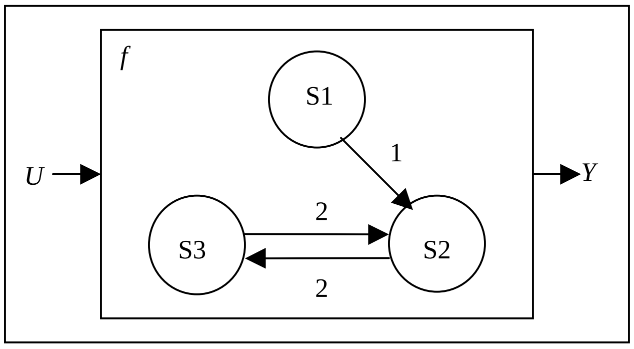


Fig. 2. Example FSM

- Transitions in our FSM are represented by water pipes. Water flows from one tank to another over a pipe that connects the two tanks to represent the activity a transition. Since our FSM has three transitions, we will need three water pipes. Each water pipe will correspond to a specific transition in our FSM.
- The "data transfer token" implicit in our FSM is represented by water. There is only one token, so there is a constant volume of water to correspond to the token. This volume is just enough to fill one tank of water, and all tanks should have the same volume.

Similarly, the mappings between the gazebo/walkway metaphor and the FSM are simple:

- Gazebos represent states in our FSM. When the person is in a gazebo, our FSM is in the state represented by that gazebo. Since there is only one person, only one state is active at a time. In addition, since our FSM has three states, we will need three gazebos. Each gazebo will correspond to a specific state in our FSM.
- Transitions in our FSM are represented by walkways. The person moves from one gazebo to another over walkways that connect the two gazebos to represent the activity of a transition. Since our FSM has three transitions, we will need three walkways. Each walkway will correspond to specific transition in our FSM.
- The "data transfer token" implicit in our FSM is represented by the person.

Step 5: Create Model
Here is a basic outline of the steps involved in creating the models:

1. Specify the basic FSM components, such as state and transition. Alternatively, take these components from a library. Place these in the *model file*. Note: these components should be generic and be as non-specific as possible to the model currently under construction.
2. Specify the FSM topology. That is, specify the number of states in the FSM, and the arrangement of transitions that exist between the states. Alternatively, take this topology from a library. Place this topology in the *model file*.

3. Either by hand or with a third-party geometry-modeling tool, create geometric 3-D analogs for each component of the FSM that will be visualized. Alternatively, take these objects from a library. In the case of our FSM, we could create cylindrical water tanks or gazebos to represent states, pipes or walkways to represent transitions, and water or a person to represent the data transfer token. Place these objects in the *scene file*.
4. Specify the animation behavior of the graphical components created in the previous step. Alternatively, take these behaviors from a library. An example behavior for water tanks is the "filling" or "emptying" of the tanks with water. An example behavior for the walkways is the movement of the person over the walkways. Place these behaviors in the *model file*.
5. Merge the *scene* and *model* files with the Model Fusion Engine's GUI interface.

The implemented water tank world, authored by Donahue [16] is shown in Fig. 3. The implemented gazebo world, authored by Kohareswaran [17] is shown in Fig. 4.

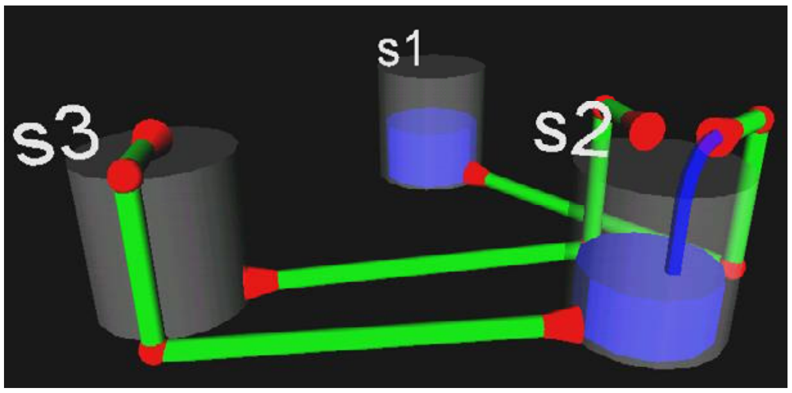


Fig. 3. Example FSM with water tank metaphor applied. Author: R. M. Donahue.

4 Summary

4.1 Benefits of the *rube* Framework

The *rube* framework has the potential to make building 3-D visualizations easier than is possible with other software visualization systems in direct proportion to: 1) the use of libraries of prefabricated geometric models, 2) the expressive power of VRML, X3D, and XML and 3) the ease of use of third-party geometry modeling tools. The issue of whether or not *rube* enables the creation of more "effective" visualizations depends heavily on the programmer/animator and the viewer. If the programmer/animator is the viewer, then benefits may be derived from self-construction. If the viewer is not the author, then benefits may only exist when the author has used "obvious" and/or traditional representation methods, provided extensive text-based explanations, or provided interactive controls for the viewer

These speculations are based on research mentioned in the introduction to this paper [1, 2].

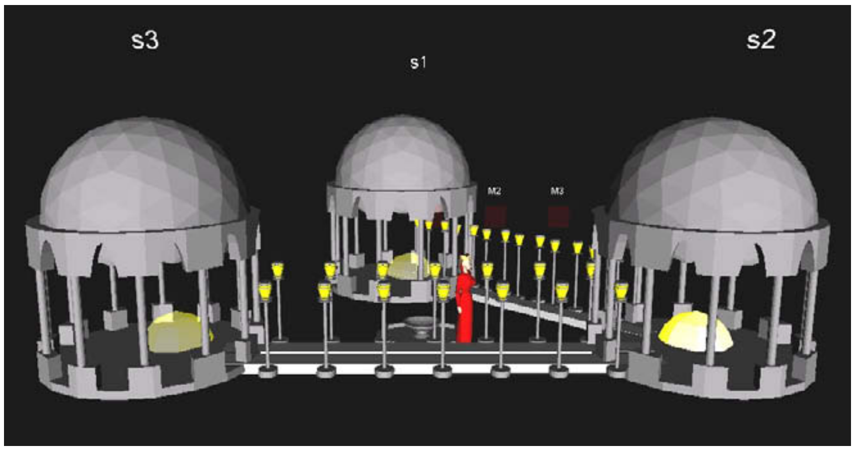


Fig. 4. Example FSM with gazebo metaphor applied. Author: N. Kohareswaran

The *rube* framework's contributions to the field of software visualization are directly related to its progress toward the first two of its previously stated goals Specifically:

1. To create a model design methodology and a software system that supports a separation of dynamic model specification from presentation and visualization.
2. To work with the Fine Arts community in creating more personalized and aesthetic presentations. The *rube* framework supports this effort by promoting the integration of modeling with developer-defined visual and audible elements.

The use of 3-D, metaphor-based visualization lends *rube* models aesthetic and artistic aspects that are relatively novel in the realm of SV, and it would be a notable achievement to blend SV with the creation of a work of art through the vehicle of 3-D metaphor. An alternative course is to limit model visualization to the creation of diagrams composed of abstract geometric shapes and symbolic text. In essence, these shapes and text are wholly arbitrary forms of representation. As such, they generally lack intrinsic semantic content. Additionally, the generic nature of these shapes and text lessens their potential aesthetic impact. These shortcomings may be ameliorated if a developer is allowed and encouraged, by a framework like *rube*, to embellish these generic entities with customized, 3-D, metaphor-based visualizations.

We currently monitor related human-focused empirical research, and it is our ultimate aim to generate empirical research centered on the use of the *rube* development environment and modeling methodology. Before this sort of undertaking, it is necessary to have an approximation of both the environment and the methodology. Specifically, we wish to avoid a "catch 22" situation where experiments cannot be executed without entanglement of experimental results with issues related to tool quality, and quality tools cannot be constructed without solid empirical results as a design guide. We are still in the "exploring" and "engineering" phases of our research. Although this chapter presents some preliminary results of our effort in *rube's* development environment and modeling methodology, more work is needed before we can reasonably proceed with human-based empirical research. We have very recently conducted a survey, with general discussion, of aesthetic methods within a class on Modeling and Computer Simulation. The results from this survey are not yet available at the time of this writing, but will be made available in the near future

In this chapter, we have provided an overview of the *rube* framework as well as provided example worlds. The emphasis for *rube* is to permit modelers greater freedom in building their own personalized software visualizations. We briefly described a web-based graphical GUI that allows to users to merge 3-D geometry, an XML model file, and pre-existing behaviors for animating and simulating dynamic software models. Possible benefits of the *rube* framework and future directions were discussed. As software engineering further leverages modeling, our research may help in the mainstream future definition of 3-D software visualization.

References

1. A. F. Blackwell, *Metaphor in Diagrams* (Ph.D. dissertation, Darwin College, Univ. of Cambridge, Cambridge, UK, 1998).
2. C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, *A Meta-Study of Algorithm Visualization Effectiveness* (Journal of Vis. Lang. and Comp., in press).
3. P. A. Fishwick, *rube* (http://www.cise.ufl.edu/~fishwick/*rube*/intro/index.html, 2001).
4. P. A. Fishwick, *Aesthetic Programming* (Leonardo magazine, MIT Press, Cambridge, MA, to be published 2002).
5. R. M. Cubert, T. Goktekin and P. A. Fishwick, *MOOSE: Architecture of an Object Oriented Multimodeling Simulation System* (Proc. Enabling Technology for Sim. Sci., SPIE AeroSense 1997 Conf., Orlando, FL, USA, April 22-24, 1997) pp. 78-88.
6. P. A. Fishwick, *SIMPACK: Getting Started with Simulation Programming in C and C++* (1992 Winter Sim. Conf. Proc., Arlington, VA, USA, December, 1992), pp. 154-162.
7. K. Lee and P. A. Fishwick, *OOPM/RT: A Multimodeling Methodology for Real-Time Simulation* (ACM Trans. on Modeling and Comp. Sim., **9**(2), 1999), pp. 141-170.
8. P. A. Fishwick, *Simulation Model Design and Execution* (Prentice-Hall, Englewood Cliffs, NJ, 1995), 448 pp.
9. P. A. Fishwick, N. H. Narayanan, J. Sticklen and A. Bonarini, *A Multi-Model Approach to Reasoning and Simulation* (IEEE Trans. on Syst., Man and Cybern., **24**(10), 1992), pp. 1433-1449.
10. P. A. Fishwick and B. P. Zeigler, *A Multimodel Methodology for Qualitative Model Engineering* (ACM Trans. on Modeling and Comp. Sim., **2**(1), 1992), pp. 52-81.
11. T. Kim and P. A. Fishwick, *A 3D XML-Based Visualization Framework for Dynamic Models* (2002 Web3D Conference in Monterey, CA, submitted 2001).
12. M. Najork, *Programming in Three Dimensions* (Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, 1994).
13. T. Kim and P. A. Fishwick, *Virtual Reality Modeling Language Templates for Dynamic Model Construction* (Enabling Technology for Sim. Sci., SPIE '01 AeroSense Conference, Orlando, FL April 2001).

14. G. C. Roman and K. C. Cox, *A Taxonomy of Program Visualization Systems* (IEEE Computer, **26**(12), 1993), pp. 11-24.
15. M. A. Najork and M. H. Brown, *Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System* (IEEE Trans. on Vis. and Comp. Graphics, **1**(2), 1995), pp. 175-193.
16. R. M. Donahue, *Industrial Plant Metaphor with a Change in Geometry* (http://www.cise.ufl.edu/~fishwick/*rube*/tutorial/Fsm4/world4.wrl, 2001).
17. N. Kohareswaran, *World with a Human Agent Metaphor* (http://www.cise.ufl.edu/~fishwick/*rube*/tutorial/Fsm5/world5.wrl, 2001).

Algorithm Explanation: Visualizing Abstract States and Invariants

Reinhard Wilhelm[1][1], Tomasz Müldner[2], and Raimund Seidel[1]

[1] Informatik, Universität des Saarlandes,
66123 Saarbrücken, Germany
{wilhelm,rseidel}@cs.uni-sb.de
[2] Jodrey School of Computer Science,
Acadia University,
Wolfville B0P 1X0 N.S. Canada
tomasz.muldner@acadiau.ca

Abstract.

Software visualization provides methods to facilitate the understanding of algorithms and programs. Practically all existing visualization systems actually execute the code to be visualized on sample input data. In this paper, we propose a novel approach to algorithm explanation based on static program analysis, specifically shape analysis. Shape analysis of a program operating on heap-based data structures analyzes the program to find out relevant properties of its heap contents. The shape analysis we are using is parameterized with sets of observation properties, which are relevant properties of heap elements. Shape analysis associates sets of shape graphs with program points. These graphs describe both structural properties and non-structural properties such as sortedness. By summarizing sets of undistinguishable heap cells shape analysis supports focusing on active parts of the data structure. By computing the program's invariants it provides the basis for their visualization. After the application of the shape analysis the program is visually abstractly executed, i.e., traversed with a strategy corresponding to a meaningful explanation. Showing a sequence of shape graphs produced along a program path demonstrates how invariants are temporarily violated and then restored.

1 Introduction

Software Visualization is the representation of information about software by means of computer graphics. There are two kinds of information that one may wish to visualize. The first is *static* information, such as the module structure of a software system, the class hierarchy of an object-oriented system, the call graph of a module, or the control flow graph of a procedure. The second is *dynamic* information available during the execution of a program, such as the current state of the program's memory.

Algorithm Animation as an aid for improved algorithm understanding typically relies on such dynamic information. The approach described in this paper attempts to achieve algorithm animation and understanding by using the information most important to algorithm understanding, namely invariants. An invariant at a program

[1] supported by the Deutsche Forschungsgemeinschaft, Project Ganimal

point describes properties of all the states the program may be in, when execution reaches this program point. It is thus a static description of a dynamically arising set of states. Our approach is unique in that the invariants and thereby the basis for our animation are statically computed employing the method of abstract interpretation developed in the context of compiler design. There is some similarity between our approach and a declarative method of algorithm animation, in which one can select a set of properties characterizing the data, and use these data to formulate and then visualize the invariants. (For the description of Leonardo, which offers a declarative language to specify properties of execution states, see [9]. However, the traditional declarative visualization is unable to analyze data on the basis of abstract rather then concrete program execution.

The construction and the working of most traditional algorithm animation systems can be described as follows:

1. The *designer* selects a data structure and an algorithm to animate.
2. He decides how to represent the execution state.
3. He determines *interesting events* (in the execution) of the algorithm.
4. A *user* then supplies a set of sample input data and runs the program on this input data. Each state of the execution is visualized in the designer's representation.

Most of the time, Algorithm Animation actually is Program Animation, as the algorithm is coded in an existing programming language and the code is shown along with some view onto the execution state. We actually deal with programs, but adhere to this imprecise nomenclature.

Our criticism of traditional algorithm animation is that on the one hand, they show *too much*, e.g. currently irrelevant and distracting parts of the execution state; on the other hand, they do not show *enough*, namely the most essential part of every algorithm - its invariants. Consider the insertion into a binary search tree. Typically, the entire tree is shown, although the insertion takes place in one of the subtrees, and so the other subtree is not relevant for the current situation. Unfortunately, the invariant stating that for each inner node, the elements in its left subtree have data components smaller than that of the node and elements in its right subtree have data components larger than that of the node, is not shown.

In this paper, we propose a novel approach to the explanation of programs working on heap-based data structures, for which this criticism does not apply. Our system can be made to focus on the "active" parts of the data structures and it will allow visualizing invariants. It is based on preprocessing the program by a static program analysis.

Let us identify the phases in the construction and use of an algorithm explanation system in analogy to the above description:

1. The *designer* selects a data structure and an algorithm to be explained. He then selects a set of properties characterizing the data structure, i.e., those properties that would be used to formulate invariants for the algorithm.
2. He runs a *shape analysis*, as implemented in the TVLA (Three-Valued-Logic Analyzer) system, (see http://www.math.tau.ac.il/~rumster/TVLA/) on the program using properties mentioned above. The shape analysis will annotate each program point with a set of *shape graphs* describing all heap contents that may exist when program execution reaches this program point. Together, all these shape graphs form an invariant for this program point.

3. He defines a visual representation for the shape graphs computed by the shape analysis.[1]
4. Now, comes the *user's* time! What does he get to see and how does he get to see it? The decisions made in point 1 above determine *what* and the decisions made in point 3 above determine *how*. The user may want to select one particular shape graph at the entry to one particular program statement and then execute the statement and observe the effect on the chosen shape graph. He may also want to perform a *visualized abstract execution* of the algorithm. This is a depth-first traversal of the program's control flow graph together with the sets of shape graphs at the program points. Each step in the visual execution corresponds to the execution of one statement as described above.

This paper is concerned with phase 3. Visualizing invariants is more rewarding, but also more difficult than visualizing concrete states. Actually, we never show the whole invariant at once, as it would be done in program verification. There, the invariant for a non-trivial algorithm may be a quite large logical formula, which takes a long time to parse and even longer to understand. Our hypothesis is that graphically displayed invariants are more intuitive, and that as much as possible of what they express should be visually represented.

Shape analysis is described in [1] and [2]. A system, TVLA, implementing it and thus supporting step 2 is available. In [2] the authors also show how shape analysis can be used to prove partial correctness by computing invariants.

The remainder of this paper is organized as follows. First, we describe an algorithm used in our paper as case study. Then, we briefly introduce static program analysis and in particular shape analysis. Next we present our approach to algorithm explanation; finally, we describe one related approach to algorithm explanation.

2 Case Study

We explain our approach with the example of insertion sort using a singly linked list. The basic idea of this algorithm, see [3], is to split the sequence of data to be sorted into two consecutive sub-sequences; a sorted prefix, pointed to by x, and an unsorted suffix, pointed to by r, which follows the prefix. Initially, the prefix is empty, and the suffix consists of all elements. The algorithm loops until the suffix is empty (and so the sorted prefix contains all the elements). In each step of the loop, the algorithm inserts into the prefix the element, pointed to by `r`, that immediately follows the prefix, thereby keeping the prefix sorted. Therefore, the basic invariant of this algorithm is: "the prefix is sorted". The right place to insert the element `r` is searched by the inner while loop. It runs a pair of pointers (`pl, l`) through the sorted prefix, until `l->data > r->data` holds. The element `r` is then inserted between `pl` and `l`. For our considerations, we need a concrete implementation of insertion sort. Consider a singly linked list of double values defined by:

[1] In principle, there could be different representations of shape graphs in different regions of a (large) program; see "Specifying Algorithm Visualizations: Interesting Events or State Mapping?" in this volume. This issue is not relevant for the small-program domain we are working in.

```
/* list.h */

typedef struct elem {
  struct elem* n;
  double data;
}* List;
```

The code of insertion sort is provided in Fig. 1 (in this code, we show two labels that will be used in discussions in future sections).

```
/* insertion.c */
#include "list.h"

List insert_sort(List x) {
/* sort list x, return sorted list */
  List r, pr, l, pl;
  r = x;
  pr = NULL;
  while (r != NULL) {
    /* while the suffix is not empty */
    l = x;
    pl = NULL;
    while (l != r) {
      /* P1: */ if (l->data > r->data) {
        /* P2: after positive outcome of
               comparison, move r before l */
        pr->n = r->n;
        r->n = l;
        if (pl == NULL)
           x = r;
        else
          pl->n = r;
        r = pr;
        break;
      }
      pl = l;
      l = l->n;
    }
    pr = r;
    r = r->n;
  }
  return x;
}
```

Fig. 1. Insertion Sort

3 Static Program Analysis, Shape Analysis, and Uncertainty

In this section, we provide a brief introduction to shape analysis, a particular static program analysis method, see [4]. This introduction provides enough details to explain the proposed approach to algorithm explanation; for a detailed description of shape analysis see [1].

3.1 Static Analysis

Static program analysis attempts to extract dynamic, i.e. run-time properties of a program without actually executing it. Since most interesting properties of programs are undecidable, the analysis can only be done by *approximation*. These approximations must be *conservative*, i.e., reliably answer questions about the program. Since in some cases, a static analysis does not know a more precise answer, a perfectly acceptable answer is "I don't know". Static analysis—as defined in the theory of abstract interpretation, see [5]—uses *abstract domains*, which are approximations of the *concrete domains* on which the semantics is defined. The effects of the execution of statements on abstract domain elements are described by abstract *transfer functions*, which are approximations of the concrete semantics of the statements.

3.2 Shape Analysis

Shape analysis is a specific static program analysis technique. For a given program and each point P in the program, shape analysis computes a finite, conservative description of the heap-allocated data structures that can arise when execution reaches P. This description can be as strong as the invariant needed to prove correctness of the program.

This invariant characterizes the data structures in the actual heap and is composed of properties of individual heap elements. There are two kinds of properties; structural and non-structural.

Examples of typical *structural properties* of heap elements are:

- being pointed to by a specific pointer variable
- being pointed to by a pointer component of a heap element
- being reachable from a specific pointer variable or some pointer variable
- being the target of at least two different heap pointers; and
- lying on a cycle.

Non-structural properties encompass properties such as having a data component that is greater than that of the left neighbor (child) and smaller than that of the right neighbor (child). All properties together are called *observation properties*.

In general, algorithms on linked data structures work on data structures of unbounded size. Any shape analysis has to find a *representation of bounded size* in order to guarantee termination. An abstraction function maps concrete heaps to their bounded representation. The shape analysis underlying our approach is generic and represents a whole family of instance analyses. The abstraction function of a particular instance is obtained by selecting a subset of the observation properties, the so-called *abstraction properties*. The *abstraction function* summarizes, i.e. maps to the same abstract heap element, all concrete heap elements that do not have observable differences in terms of the abstraction properties. All concrete heap elements represented by the same *abstract heap element* agree on their abstraction properties; i.e. either they all have these abstraction properties, or none of them has them. Thus, summary nodes inherit the values of the abstraction properties from the nodes they represent. For non-abstraction properties, their values are computed in the following way: if all summarized elements agree on this property, that is they have the same value, then the summary node receives this value. If not all summarized elements agree on a property, their summary node will receive the value “don’t know”. Therefore, there is a need for *three values*; two definite values, representing 0 (false), and 1 (true), and an additional value, ½, representing *uncertainty*.

The abstract heap is also called a *shape graph*, because it can be represented as a graph, whose shape provides information about data structures in the heap. Note that the set of nodes of the shape graph is determined by the set of abstraction properties; in particular it does not change if one adds a new non-abstraction property (of course, the value of this new property will have to be properly established for all abstract nodes). The edges of the shape graph are computed by abstraction from binary connected-by-pointer-component properties in a way exemplified in examples later on.

As the first, simple example of a shape graph, consider an acyclic singly linked list of length at least 3 pointed to by a pointer variable x and connected by n-pointers. Let's have as the only abstraction properties, the properties of the kind "pointed-to-by-p", for all program pointer variables p. The corresponding abstraction function will map the head of the list to one abstract element (unique node), let's call it *u*, since the head is the only list element satisfying the property to be pointed to by x. The whole tail of the list will be mapped to another abstract summary node, let's call it *v*, because they all don’t have the property of being pointed to by x. Thus the shape graph for this example consists of the two nodes; *u* and *v*.

Now, consider an additional, binary, non-abstraction property: "connected-by-n-component". This property describes the edges when the data structure is seen as a graph. No two different summarized list cells agree under this new property, since at any one time a pointer component may point to at most one heap cell. Thus, whenever we summarize several heap cells, the connected-by-n-component properties of the resulting summary nodes have the value ½. Hence all edges in this graph are “uncertain” edges, see Fig. 2.

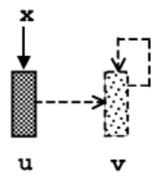


Fig. 2. Shape graph for a singly linked list

In Figures 2, 3, and 5, which are based on the output from the TVLA system, the following conventions are used: summary nodes appear in dotted rectangles, and unique nodes appear in solid rectangles. Edges are solid if the value of the property is 1, and they are dotted if the value is ½. Program pointer variables are shown at the top, with vertical arrows with solid tips indicating their values. Node names used in the discussion in the paper are shown as labels of nodes.

For a second example of a shape graph, consider the Insertion Sort procedure of Fig. 1, which has five pointer variables x, pl, l, pr, and r. They induce the five "is pointed to" abstraction properties *x(v)*, *pl(v)*, *l(v)*, *pr(v)*, and *r(v)*. Now consider the reachability property $r_z(v)$, whether *v* is reachable from pointer variable z, and use

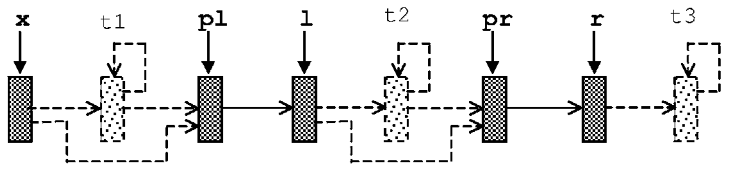


Fig. 3. Shape graph for the insertion sort procedure

$r_x(v)$, $r_{pl}(v)$, $r_l(v)$, $r_{pr}(v)$, $r_r(v)$ as five additional abstraction properties. One of the shape graphs that can arise during the sort procedure of Fig. 1 is depicted in Fig. 3.

Note that the reachability properties are crucial for obtaining such an abstraction. The nodes represented by `t1` are (only) reachable from `x`, the ones represented by `t2` are also reachable from `pl` and `l`, and those represented by `t3` are in addition reachable from `pr` and `r`. Since all these nodes have the reachability properties, we can reinterpret the fact that several `n`-edges are uncertain, i.e., have value ½. Before, we would interpret *$n(v,w)$ = ½* for a summary node w as "the `n`-component of the heap cell represented by *v* may point to one of the heap cells represented by *w*". Because of the reachability information we can say, "the `n`-component of the heap cell represented by *v* points to one of the heap cells represented by *w*". The `n`-edge bypassing `t1` represents the fact that this sublist may in fact be empty.

4 Algorithm Explanation

In this section, we describe our approach to algorithm explanation. We propose to preprocess an algorithm by running a shape analysis on this algorithm and then *visualize an abstract execution*. Instead of executing the algorithm with concrete input data and on concrete states, it is executed with a *representation of all input data* and on abstract states.

4.1 Focusing

Before we describe our proposed visualization, let's briefly recall from [6] the support provided by shape analysis for the focusing on *active* parts of the heap contents, i.e. those parts of heap data structures where the algorithm currently works. A component of the data structure is *active* at some time during execution if one of the program's pointer expressions currently points to it. The set of pointer expressions occurring in the program can be statically determined. For each pointer expression a predicate is specified, which holds for heap cells accessed by the pointer expression. The abstraction function instantiated with this set of predicates will uniquely represent the accessible heap cells and summarize the rest. The set of unique nodes is then called the *focus*. This focus may still be too large for a point in the program since it is determined syntactically from the whole program. A simple preprocessing may, however, suffice to narrow down on the locally active parts.

In this paper the issue of focusing will not be further pursued.

4.2 Visualization of Non-structural Properties

This paper deals with algorithms on totally ordered domains. Our particular example is an insertion sort, in which the order, *dle*, between data structure components is induced by the total order on the data domain. The set of properties necessary to analyze Insertion Sort is listed next. We have already seen a partition of the set of properties into abstraction and non-abstraction properties. Now, we meet a different partition, into structural and non-structural properties. The structural properties concern the connectivity in the data structure. The examples shown above used the following structural properties:

x(v)	is *v* pointed to by pointer variable x
n(v1,v2)	does the *n*-component of *v1* point to v2
$r_x(v)$	is *v* reachable from pointer variable x

The additional, non-structural, predicates are:

dle(v1,v2)	is the data-component of *v1* less than or equal to the data-component of *v2*
$inOrder_{dle}(v)$	does *v* have no n-successor or is the data-component of *v* less than equal to that of its n-successor.

Traditional algorithm animations map the sizes of elements into a different, *visualizable* totally ordered domain, such as the height of the object representations. However, the large number of such objects distributed all over the screen may be easily confusing; for example, if there are five or more objects of different heights. Therefore, we are using a different representation of the order, by mapping these values to positions on the x-axis. As our examples will demonstrate, it is much easier to understand the relationships between elements placed on the x-axis than having to compare their heights. A particular problem associated with our approach to visualization is the necessity to deal with the lack of information about data values. Let us try to show this by comparing how a sorting program works and how the shape analysis of the sorting program works.

Once the input to the sorting program is given, the sorted permutation of this input is uniquely determined (up to the placement of repetitions). The task of the sorting program is it to compute the permutation. Its execution state "increases in sortedness" by executing comparisons and reordering elements correspondingly.

Shape analysis applied to this sorting program starts with a description of all possible inputs and has to compute invariants at all program points, in particular show that the postcondition holds, namely that upon termination the elements are sorted. Invariants do not refer to actual data, but only mention relative sizes. For example, an invariant for a sorted sublist could be that the data-component of each but the last element of the list is less than or equal to the data-component of its successor. This is expressed by the fact that the *inOrder* property holds for the node summarizing this sublist. Shape analysis abstracts from the values of data-components. Data structure elements are thus incomparable to start with, but "gain comparability" by abstractly executing conditions. Mathematically, we would say that only a *partial order* on the elements of the data structure is known to the shape analysis. Abstractly executing comparisons may insert elements into chains, i.e. totally ordered subsets of the partial order.

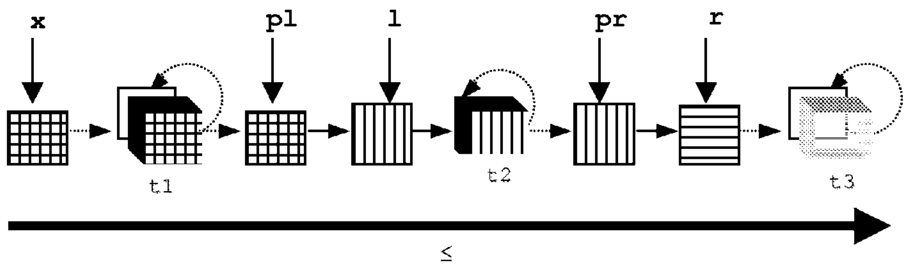


Fig. 4. A shape graph for a partially sorted linked list during insertion sort (at point P1)

Consider the insertion sort algorithm given in Fig. 1. Assume the inner loop has been executed several times and that the current execution point is P1, i.e. just before evaluating the condition `l->data > r->data`. One abstract heap at this point is shown in Fig. 4. Horizontal arrows are implicitly labelled *n*. Unique nodes are shown as rectangles and summary nodes as cubes; non-empty lists of nodes are shown as cubes with an additional rectangle at the back. Dashed self-loops show that we deal with summarized sublists, i.e. sets of heap cells connected by *n*-components. Finally, the placement on the x-axis indicates the relative size of the represented concrete elements.The graph in Fig. 4 represents all lists with the following properties:

- pointer variable `x` points to the head of the list
- the head is followed by a non-empty sublist, in the figure represented by a summary node called `t1`
- the element following this sublist is pointed to by pointer variable `pl`
- `pl`'s `n`-component points to an element pointed to by pointer variable `l`
- the element pointed to by `l` is followed by a possibly empty sublist, summarized by `t2`, which is followed by an element pointed to by `pr`
- the prefix of the list up to the element pointed to by `pr` is already sorted, we therefore call it the *sorted prefix*
- the next element in the list is pointed to by pointer variable `r`; it is the one to be inserted in the sorted prefix; the suffix of the list following this element is represented by `t3`
- the prefix of the list up to the one pointed to by `pl` have already been compared to the element pointed to by `r`; we call the prefix they form the *compared prefix.*

This intuitive description is in fact part of the invariant for this program point. It contains a “structural component” describing an infinite set of singly linked lists with a set of pointers pointing into this list and some conditions on the minimal lengths of sublists. It also shows sortedness and comparability properties of the represented list elements and sublists.

4.3 Visualization of Uncertainty

There is a complication in using the above representation introduced by the fact that some elements may be incomparable. Shape nodes are placed left-to-right according to the *dle*-order. Looking at Fig. 4 one could get the impression that all elements shown in this figure are comparable with respect to *dle*. This is not the case, for example, the element pointed to by `r` is incomparable with any element in the suffix of the compared prefix starting at the element pointed to by `l`, since it has not been compared with them. Therefore, we need a graphical representation of partial order on shape nodes avoiding the impression of comparability between elements that are in fact incomparable. For this reason, we consider ascending chains. The left-to-right placement is then only relevant for nodes that are members of the same chain. We define binary predicate *inChain* in the abstract domain as follows:

inChain(v, w) iff the following three conditions hold:
if *n(v, w)* != 0, then *inOrder(v)* = 1
if *n(v, w)* = 0, then *dle(v, w)* = 1
if *w* is a summary node, then *inOrder(w)* = 1

An ascending chain is either a single node *v* with *inOrder(v)* = 1 in case *v* is a summary node, or it is a sequence $(v_1,...,v_n)$ such that v_i *inChain* v_{i+1} for i=1, 2, ...,n.

As examples consider the two chains:

```
x, t1, pl, l, t2, pr
x, t1, pl, r
```

in Fig. 4. In the following, we demonstrate two different ways to represent chains. The first represents chains by patterns or colors; the combination of two or more patterns or colors indicates that a node belongs to more than one chain. For example, the first of our example chains in Fig. 4 may be indicated by vertically striped nodes, the second chain may be indicated by horizontally striped nodes (nodes in both chains exhibit both kinds of stripes). The second representation will be explained in Section 4.5.

4.4 A Transition

We will now describe what happens in the transition corresponding to the comparison "`l->data > r->data`", assuming it is true. Fig. 6 shows the next state of the algorithm, when the current program point is P2 (see Fig. 1). At this point, we have the information, that "`l->data > r->data`". This transition increases the sortedness of the list as now the element to be inserted has found its place between the `pl`- and the `l`-elements.

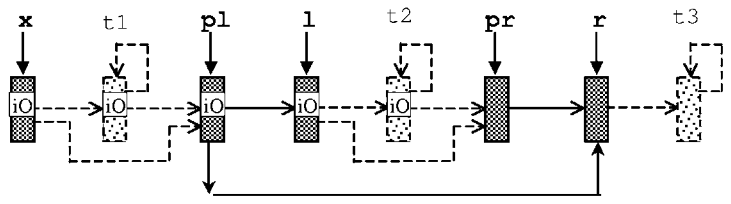


Fig. 5. Shape graph for Insertion Sort; as in Fig. 4; only one dle-edge needed to construct the chains is shown. Nodes with the "iO" inside have the inOrder property.

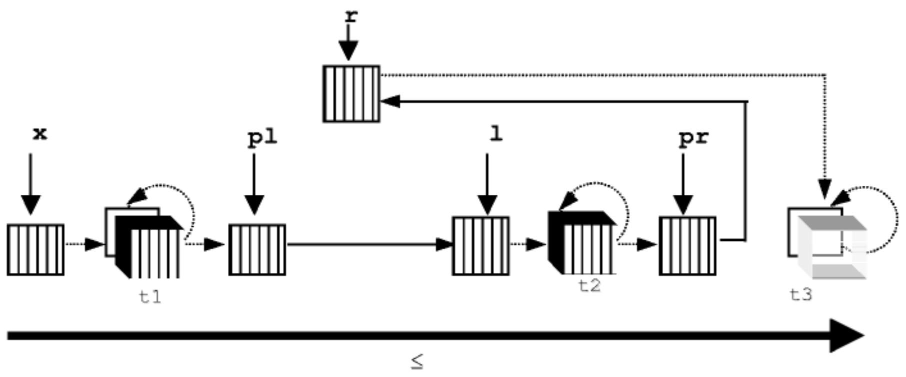


Fig. 6. A shape graph for a partially sorted linked list during insertion sort (at point P2, i.e. after the comparison with positive outcome).

Therefore, the element pointed to by `r` appears now between these elements**.** There is only one chain, representing the sorted prefix.

4.5 A Different View

Here we provide a different view to display the structural and non-structural properties of shape graphs in the context of sorting linked lists (recall their definitions from the beginning of Section 4.2). In this visualization the structural properties are represented just as described for shape graphs in Section 3.2. Non-structural properties are implicitly represented by the placement of the boxes relative to each other, rather than by explicit attributes, such as colors, used in the previous visualization. Now the non-structural property *dle(v1,v2)* is implicitly represented by the fact that there is a horizontal line that intersects the boxes corresponding to *v1* and *v2*, moreover *v1*'s box is to the left of *v2*'s box. The property $inOrder_{dle}(v)$ will be shown by the fact that the `n`-arrow leaving *v*'s box emanates from the right border of that box. The case that property $inOrder_{dle}(v)$ does not hold will be indicated by the `n`-arrow leaving *v*'s box from the left border (e.g. the node pointed to by `pr` in Figure 8), and if it is unknown whether the property holds the `n`-arrow emanates from the top or bottom side of the box. The *inOrder* information is not explicitly represented in our first visualization, but has to be deduced from the fact that the summary node is member of a chain. Chains are now represented as follows: if there is a horizontal line that stabs in left to right order first v_1's box, next v_2's box, and so on up to v_n's box, then $(v_1,v_2,...,v_n)$ is an ascending chain. In Figures 7 and 8 we show this alternative representation for the same situations depicted in Figures 4 and 6. This way of implicitly representing order information is quite versatile, especially if rectangles of different heights are used for the node representations. However, there are some intrinsic limitations in that only a limited number of interacting ordered chains can be represented. These limitation are a consequence of Helly's Theorem, see [8].

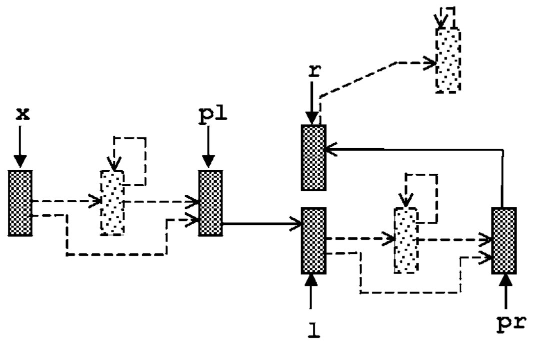


Fig. 7. Visualization from Fig. 4 with different representations of chains

4.6 A Résumé of Our Approach to Algorithm Explanation

Let us repeat the description of the construction process of an algorithm explanation in slightly refined form:

1. Select a data structure and an algorithm to be explained. (In this article, it is Insertion Sort implemented on linked lists.) Select the right set of observation properties characterizing the data structure. In our setting, we split the set of observation properties into subsets of structural and non-structural properties, and a subset of all observation properties, the set of abstraction properties.
2. Run the system, TVLA, implementing the generic shape analysis instantiated with these sets of predicates on the program. It will annotate each program point with a set of shape graphs.
3. Define visualizations for abstract states, in particular of (the states of) the data structures. Visual representations for nodes in the shape graphs have to be defined, in particular representations for unique nodes and for summary nodes. Also, graphical or geometric ways for visualizing chains have to be chosen.

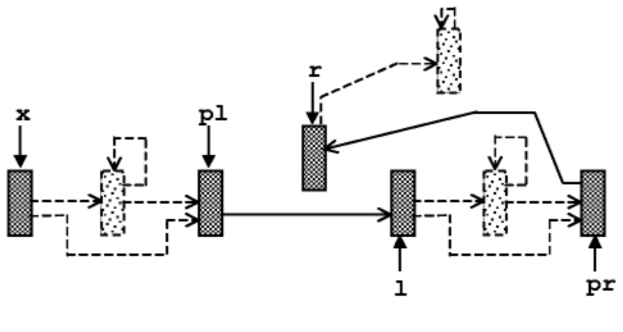


Fig. 8. Visualization from Fig. 6 with different representations of chains

4. Process the analysis output, i.e. the shape graphs associated with the program points, according to the chosen visualization. In our case, this means determining the graph layout, computing the ascending chains covering the partial order and visually represent them.

5 Related Work

We deliberately do not compare our work to the vast literature on traditional Algorithm Animation. The difference has been made sufficiently clear in the Introduction. The corresponding literature will be discussed elsewhere in this volume.

To our knowledge, the only approach similar to ours has been tried by Amir Michail at the University of Washington see [7], who attempts to provide algorithm explanation by programming in a visual programming language. His work was mentioned to us after a talk at the Dagstuhl Seminar on Software Visualization; see http://www.dagstuhl.de/DATA/Reports/01211/, and we briefly describe it in this section. Michail's approach has been exemplified using Binary Tree Algorithms. There are two important concepts in his approach: tree fragments and abstract nodes. *Tree fragments*, visualized as triangles, correspond to our summary nodes and represent connected subgraphs of binary trees (note that tree fragments do not need to form subtrees). Abstract *nodes*, visualized by circles, always uniquely represent individual concrete nodes. Nodes and tree fragments may bear colors to show respectively properties of the node or the sets of nodes. While for a node the color signals definite satisfaction of the property, for tree fragments, the properties may existentially or universally hold (it is not clear from the visualization which of these two properties holds, and additional specification is necessary to clarify this). A multi-colored tree fragment indicates that different properties hold for disjoint subsets of the represented nodes. There is no explicit and precise connection between graph components and colors, for example, in the case of a binary search tree, there is no association of the left subtree with the property "to be less than a key", and for the right subtree "to be greater than the key".

The border between differently colored regions, a *path*, has special significance; for instance, it may be a search path for a given key. A path has no color and so it is unclear which properties nodes on this path will have.

The absence of a color means that the tree fragment contains no element with this property (or maybe that not all elements have this property?). Many additional properties have to be textually described, for example the BST property. It seems that in the absence of a semantics foundation, such as shape analysis, this approach produced many ad hoc decisions and consequently introduced many inconsistencies.

6 Conclusion

A novel approach to algorithm explanation based on showing invariants was presented. The invariants are pre-computed by shape analysis, a powerful static program analysis method. Using shape analysis is a first step towards the automatic creation of algorithm animations. Further steps are needed. The number of shape graphs may be far too large to be traversed by a viewer, because shape analysis exhaustively generates all cases the algorithm may encounter. A human explainer would be able only treat a small number of those cases to explain the algorithm (and by accident may forget some very relevant cases). We foresee ways to reduce the number of cases without loosing explanatory quality. Appropriate visualizations of structural properties have to be found and visualizations of non-structural properties have to be added on top of them. A visual calculus is envisioned in which the results of the shape analysis of a piece of software can be visualized more or less automatically.

We encountered this criticism: "You talk about automation and then you visualize a single program." However, the specification for a data structure, e.g. a linked list, is useful for many programs working on linked lists. If additional properties are required, say sortedness, only the corresponding predicates have to be supplied. An even stronger counterargument is that the complete specification of our running example was recycled from the (successful) attempt to verify partial correctness using shape analysis, cf. [2], with no additional work required! This will be so in many cases, since verification involves invariants, which sometimes can be computed by shape analysis, and invariants is what we want to visualize.

The state of implementation of our approach is the following: The underlying shape analysis is implemented in the TVLA system. Included in this distribution is an traversal strategy designed to visually execute the result of the analysis. TVLA offers a graphical output format using Graphviz. We are currently experimenting with more adequate visualizations such as the ones in this paper.

References

1. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape-analysis. In: Proc. of ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, St. Antonio, Texas, (1999), extended version to appear in ACM Transactions on Programming Languages and Systems (TOPLAS).
2. Lev-Ami, T., Reps, T., Mooly, S., Wilhelm, R.: Putting static analysis to work for verification: A case study. Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, USA (2000) 26-38
3. Aho, A.V., Hopcroft, J.E., Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Mass. (1983)
4. Nielson, F., Riis Nielson, H., Hankin, C.: Principles of Program Analysis. Springer-Verlag, (1999)
5. Cousot, P., Cousot R.: Systematic design of program analysis frameworks. Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, U.S.A., San Antonio, Texas, (1979) 269-282
6. Braune, B., Wilhelm, R.: Focusing in algorithm explanation. Transactions on Visualization and Computer Graphics, 6(1), (2000) 1-7
7. Michail, A.: Teaching Binary Tree Algorithms though Visual Programming. In Symposium on Visual Languages, IEEE, (1996) 38-45
8. Wenger, R.: Helly-Type Theorems and Geometric Transversals. In J.E.Goodman, J. O'Rourke (Eds.): Handbook of Discrete and Computational Geometry, CRC Press, (1997)
9. Demetrescu C., Finocchi, I. and J. Stasko.: Specifying Algorithm Visualizations: Interesting Events or State Mapping? In this volume, p. 105.

Visualisation and Debugging of Decentralised Information Ecosystems

Rolf Hendrik van Lengen and Jan-Thies Bähr

German Research Centre for Artificial Intelligence
Intelligent Visualisation and Simulation Systems Department
Erwin-Schrödinger-Str. 1,
D-67608 Kaiserslautern, Germany
{lengen, baehr}@dfki.de
http://www.dfki.de

Abstract.

The multi-agent software platform DIET[1] provides components to implement open, robust, adaptive and scalable ecosystem inspired applications. The development, debugging and monitoring of decentralised systems is a complex process. Adequate techniques and tools are required to assist the application developer. This paper describes the outline of a platform for visualising ecosystem inspired applications. The architecture provides fundamental components and functions extending the DIET core layer by visual and interactive components. These will allow the introspection and manipulation of applications built upon DIET without the need for changing any application code.

1 Introduction

The increasing complexity of the current global information infrastructure (e.g. the Internet) requires novel means of understanding and efficiently analysing the dynamics of information. The formal concept of a Universal Information Ecosystem (UIE) was introduced by the European Commission and may become one feasible way to achieve this [1]. In a natural ecosystem material and energy are in a constant interrelation. According to the UIE proposal the information flow between many interacting software agents[2] can be regarded similar to energy in a so-called *information ecosystem*. In the same way to a natural ecosystem it is dynamic, noisy, to some extent unpredictable, and decentralised [2].

In such a networked society, information providers and consumers frequently interact, cooperate and compete with each other, while the surrounding environment is in a constant flux due to progressive commercial, political, social and technological developments. In this scenario the exchange of information is on a scale far greater than can be handled by an individual in an appropriate time period. Information

[1] Decentralised Information Ecosystem Technologies
[2] so-called infohabitants

ecosystems ideas can improve our understanding of information infrastructures. One feasible way to complete particular tasks is the implementation of ecologically inspired interactions in computational architectures.

The development, debugging and monitoring of ecosystem inspired applications is a complex process. Adequate techniques and tools are required to assist the application developer. This paper describes the outline of a visualisation architecture that extends the multi-agent software platform DIET by supplying visual and interactive components. The main purpose of this visualisation platform is the inspection and manipulation of applications built upon the DIET software platform. Special emphasis has been laid on a flexible visualisation architecture that has minimal effects on the observed application.

Section 2 gives a brief description of the DIET platform. Section 3 describes the proposed visualisation architecture and explains how monitoring and analysis of DIET applications is achieved. Section 4 illustrates some applications and their visualisation. This is followed by a discussion about related work. Section 6 provides a brief indication of further work.

2 The DIET Platform

The DIET project [3] focuses on the development of a lightweight multi-agent platform that serves to implement complex information management tasks in a real-life context. Following the design philosophy of the project the DIET platform is implemented in a decentralised, distributed, and ecologically inspired way. This design approach supports the development of open, robust, adaptive and scalable DIET applications.

The DIET software platform is designed based on concepts and techniques inspired from natural ecosystems. The platform consists of one or more environments, which accommodate large numbers of lightweight software agents, so-called infohabitants. They share common environmental resources and are capable of communicating with each other to carry out system functions and collectively support applications. DIET software agents are designed to accomplish very simple behaviours and can react rapidly to new information with minimal processing.

The architecture of the DIET software platform is layered, incorporating modularity that allows for the flexible extension of the framework. The kernel of the DIET software resides in the bottom layer, the *core layer*. It provides the fundamental functionality available to all implementations in the DIET architecture, but also embodies the constraints under which all DIET software agents must operate. Based on the *Observer Design Pattern* [4] there are visualisation hooks to the basic DIET objects that allow the observation of these objects in a non-intrusive and only loosely coupled way. This is done in a publish-subscribe manner that allows the subscribed observer to keep track of all changes of the internal state that are published by the observed subject. The *application reusable component layer* (ARC layer) includes optional components that are useful to various applications. It also contains general filtering and viewing components that allow for the visualisation and debugging of DIET applications. The *application layer* is the third layer and contains application-specific code. Associated with this layer may be specific visualisation components, to enable the visualisation of applications developed using the DIET platform [5].

3 The DIET Visualisation Architecture

In recent years, there has been a great deal of interest in visualisation and debugging of traditional distributed systems. As a result, visualisation tools for parallel debugging, performance evaluation and program visualisation have been developed [6].

Furthermore several agent-building toolkits like DCM [7], ZEUS [8][9] and AMS [10] have provided agent development environments together with basic visualisation support. These agent-building frameworks have substantially improved the study and development of collaborative multi-agent systems. Regarding visualisation these approaches, however, have several limitations not suitable for the DIET architecture.

In order to work properly, the visualisation components are realised as agent systems themselves using a certain message protocol to communicate with a heavyweight multi-agent system to be monitored and manipulated. By contrast, the DIET approach favours the design of lightweight agents that lack the implementation of a special message protocol for the sake of openness as well as a minimal core implementation allowing the creation of a large number of agents with very little overhead.

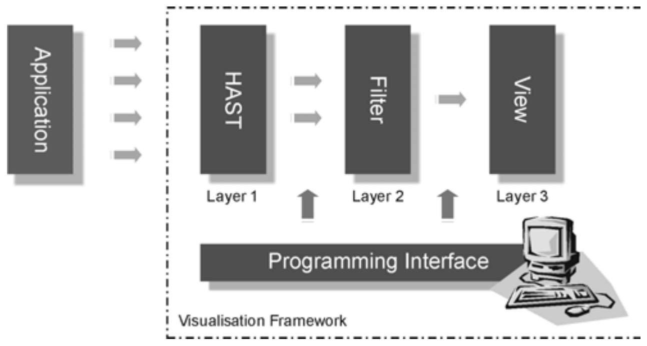


Fig. 1. DIET visualisation platform. Events (arrows) are created by the DIET application and processed by the visualisation platform in order to analyse the application.

As an attempt to address some of the limitations in recent agent-building toolkits, the DIET visualisation architecture is designed as a distinct module. DIET agents have no complex communication protocols that can be analysed and used for visualisation purposes. Instead, the DIET core software provides a basic event protocol. Events relate to the creation and destruction of elements, modification of element state, and interactions between elements in the DIET application and establish a hook to the visualisation platform. In this way, the visualisation can act as a non-intrusive observer. As a result, the design of the architecture enables the introspection of applications without the need for changing any application code.

The DIET visualisation platform is constructed as a three-layer architecture, in order to improve the visualisation performance and to minimize resulting side effects. A visual programming interface supports the creation of user defined visualisation networks. Figure 1 shows the configuration of the DIET visualisation platform. A brief description of the layered concept will be given in the following sections.

3.1 Hierarchical Application State Tracker

The DIET core layer provides the fundamental components and functions that are common to all implementations in the platform. In particular, two indispensable components of DIET are present here, *environment* and *infohabitant.* An Environment provides a location for infohabitants to inhabit and a number of services to secure infohabitant activities. An Environment is also responsible for creating and destroying infohabitants when necessary. The concept of *environment* can be extended to a *world* when DIET runs with infohabitants in multiple environments. A *world* represents a collection of environments running on the same machine.

In the DIET platform, infohabitants have no direct reference to their environment or each other. Instead, the core provides special mechanisms, which ensure that all infohabitants obey the constraints of the system in their activities. These mechanisms also support extensive “hooks” to be placed in the core to observe the interactions between infohabitants and their environment and between pairs of infohabitants. These hooks are the basis for building the visualisation and debugging platform.

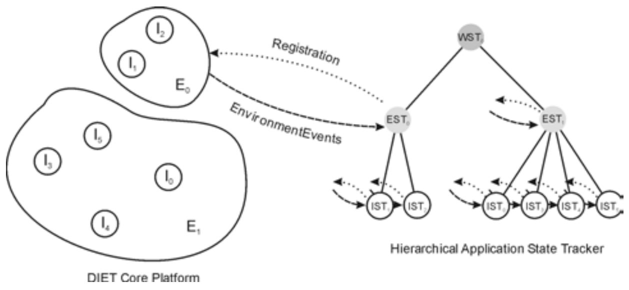


Fig. 2. The Hierarchical Application State Tracker. The figure illustrates a simple example with two environment state trackers (EST_0, EST_1) and six infohabitant state trackers (IST_0, ... , IST_5).

In the truly decentralised DIET architecture there is no central entity that can be queried about overall system properties. So it is up to the visualisation and debugging architecture to take care of such properties. The DIET visualisation platform is designed as a three-layer architecture. The visualisation process is accomplished in a publish-subscribe manner, which allows the subscribed observers to keep track of the internal state of the DIET components. The first architecture layer consists of the so-called *Hierarchical Application State Tracker* (HAST), which is organised in a similar way as the DIET world. Environment, infohabitant, communication and property state trackers are provided, each of them responsible for the corresponding component in the DIET hierarchy.

The HAST is not only listening to incoming events. The HAST reflects the internal state of the application and it will be possible to store the event history in a database for later evaluation. In addition, it aggregates data about a running application, e.g. the number of environments and the number of infohabitants in each of the state tracker as well. In addition to the *simple properties* directly defined through the application itself describing the local state of infohabitants there are *derived properties* of DIET entities that can be used to illustrate more global states of the application. For the purposes of the filtering and viewing tasks in the next layers of our visualisation architecture all these properties are treated in the same way, which makes it easier to combine, compare and compute them in a unified way to define views tailored to the specific needs of developers.

3.2 Filter Layer

Having in mind that a DIET application may consist of many more agents than conventional agent systems, the developer should be supported to handle the additional complexity. This includes the definition of filters and more sophisticated expressions derived from simpler ones to support the understanding of an application on a higher level. The developer may use his a-priori knowledge of the expected behaviour to formulate conditions and expressions that will assist him in visualising interesting information “hidden” in a possibly huge amount of events.

In order to improve the visualisation performance the user can determine which events are passed to the filtering layer of the visualisation architecture. In the filtering layer new *complex properties* can be composed out of the existing *simple* and *derived properties* the Hierarchical Application State Tracker layer offers. Event methods and their parameters (objects) can be customised by predicates. Simple and complex predicates are provided by the visualisation framework (the HAST) and by the user, respectively. A simple predicate, for example, is the `GT` (greater than) predicate. The `GT` predicate is applied to the property (e.g. resource) of an event object. It returns the Boolean value `true` if the property is greater than a pre-defined value. Complex predicates are used to concatenate simple predicates with Boolean operators and thus allow to phrase conditions for filtering or for usage as watch points to indicate certain incidents.

As a refinement step for the filtering process it is possible to manually select one or more graphical representations of DIET elements and treat this conglomeration of elements as a single element. It is also possible to feed this selection information back into the filtering process to adjust the current view. This is also an opportunity to filter elements without the need to specify explicit conditions but just to use the visual information presented to the user. While it is in some cases not so easy to define a predicate for filtering it may be more adequate for filtering larger amounts of infohabitants or to perform searches for certain property values. Hence the definition of predicates and the manual selection and grouping of DIET elements complement each other.

3.3 Viewing Layer

The third layer of the visualisation architecture consists of the viewing layer. Analogous to the filter concept a view registers as listener with the HAST hierarchy or with one or more filters. In the case of a corresponding event, the listener calls a suitable event method.

A view is a frame that contains a collection of so-called layouts. Different layouts will be provided by the platform, e.g. *Spreadsheets*, *2D-Graphs* and *3D-Graphs*. Each of these layouts contains simple graphical objects representing selected DIET components (e.g. infohabitants, properties, etc.). Simple graphical objects comprise icons, charts or textual information. The appearance of a graphical object can be customized to increase their visual perception or to meet special visualisation requirements. Out of the number of possible properties the user can choose which to map by customising the look of the graphical component, e.g. its colour, size, shape etc. In this way it is possible through the combination and customisation of graphical objects and their layout to present a variety of views to the developer.

A number of predefined standard views will serve as a starting point for inspections of the DIET world. Similar to the proposed interaction view in the Developer's Conceptual Models [7] we offer a *WorldView*. In a prototype implementation infohabitants and their communication connections are presented in a mass-and-spring model graph where infohabitants connected to each other are attracted by spring forces while other infohabitants are repelled (s. Fig. 3). Together with a *TreeView* to navigate through environments and their infohabitants these two views can be used to get an overview of all environments and infohabitants. An additional *AgentView* will show more information about individual agents. Another view presents statistical data about the application to the developer. In the context of a system containing a very large number of agents this is even more important than in most existing agent systems where the number of agents is relatively small.

Currently several pre-defined simple views are offered by the visualisation platform, e.g. the *World View* and the *Tree View*. These views are in an experimental stage and serve to test the layered concept of our architecture. They are currently not customisable and consist of a single layout only.

4 An Example

In order to show the potential of the DIET approach the platform has been applied to several illustrative tasks. The Sorting application is a simple example that tests and verifies the basic features of the DIET core layer and the visualisation platform, respectively. Figure 3 shows a snapshot of this application while it is visualised by the *WorldView*.

In this application, a set of so-called *Linker* infohabitants is sorted according the binary value of their identities. Linker infohabitants are inactive and only react to incoming messages from other infohabitants. Each Linker tries to maintain a connection to two other Linkers: both as close as possible to its own identity but one with a lower identity value and one with a higher identity value. When a Linker receives a message with the identity of another Linker, it checks if it improves either of its existing links. If it does, it updates the link and sends its own identity to the corresponding Linker. Otherwise it forwards the received identity to the link with an identity closest to it. The sorting is continuously triggered by so-called *Trigger* infohabitants. In contrast to Linkers the Triggers are active infohabitants. They trigger the Linker infohabitants to improve their links by randomly selecting two Linkers and tell one about the other's identity.

Although this application is quite simple it has a few interesting features. The sorting application is robust to failures. The result is always a perfect sorted list even when there are failed connections and rejected messages due to temporary high system load. In addition, the system is very scalable: it can handle at least 100000 infohabitants in an experimental simulation.

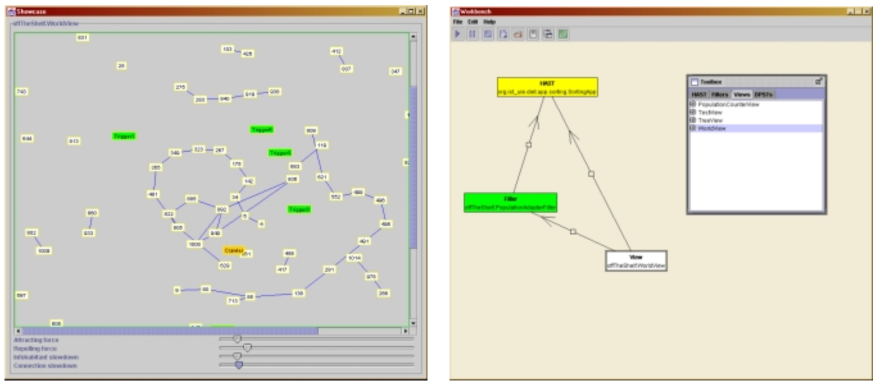


Fig. 3. Visualisation of the sorting application. Left: Individual Linker and Trigger infohabitants are shown by blocks – lines between them indicate links being formed between infohabitants. Right: A visual programming interface supports the creation of user defined visualisation networks. The network in this example consists of a HAST, a simple filter and the *WorldView*.

Visualisation of the application proved that when the sorted sequences become longer, a single trigger action leads to more system activity. This behaviour is explained by the fact that more messages are being cascaded along the sequence. The simulation showed that as the algorithm progresses all Triggers should trigger less often for optimal performance. A simple mechanism ensures this behaviour. Whenever a Trigger fails to send a message or fails to setup a connection it increases its sleep interval. At the beginning of the simulation the sleep interval is set to an initial value.

5 Related Work

In recent years, there has been a great deal of interest in visualisation and debugging of traditional distributed systems. As a result, visualisation tools for parallel debugging, performance evaluation and program visualisation have been developed [6]. Furthermore several agent-building tools are available to assist with the tasks of building distributed agent-based systems. However, visualisation and debugging these systems is a relatively new field of research in artificial intelligence.

Liedekerke and Avouris have introduced a multi-agent system development tool, based on the innovative concept of the so-called Developer's Conceptual Models (DCM). These are multiple complementary abstractions of the multi-agent system under development concentrating on specific aspects of the system. In relation to these abstractions, several system views are defined that allow a graphical representation of the selected aspects of the system state and its dynamic behaviour. An agent view to illustrate agent internal information or an interaction view to show dynamic communication are examples for these specific system views [7].

The ZEUS Agent Building Toolkit [8] [9] facilitates the rapid development of collaborative agent applications through the provision of a library of agent-level components and an environment to support the visualisation and controlling of distributed multi-agent applications. In relation to the DCM concept ZEUS offers a suite of tools, with each tool providing a different view of the application being visualised. A mailbox handles the communication between the observed application and the viewing tools. A message handler processes incoming messages received by the mailbox and delivers them to appropriate viewing tools, which have registered their interests in a context database in advance.

The Agent Management System (AMS) [10] focuses on the visualising, monitoring and analysing of in-service agents and the fulfilment of their service tasks through service level agreements. Although AMS is designed to be used in different multi-agent environments it relies completely on the co-operation with the agents to be monitored. It is designed as an agent system itself. The information management agents deal with the communication of internal and external agents and with storing events into a database. Analysis and visualisation agents perform analysis and viewing of the collected data and present it in three different views called agency level, agent level and domain level. The first one concentrates on the whole agents society and the communication between these societies. The second one describes the internal states of agents and the third presents the overall performance of the domain task.

6 Further Directions

The design of the DIET platform as an information ecosystem focuses on the development of agents that are much simpler compared to those in most existing systems, and that do not rely on centralised services, but explore their surrounding environment by the means of local communication with other infohabitants. Together with evolving and changing strategies of infohabitants there is a need to visualise such behaviour (leading to domain task and co-operation views). While it is easily possible to perceive the fact that communication between infohabitants occurs and to show its intensity it is not as easy to analyse and categorise the kind of communication. It might be of high interest to distinguish if the infohabitants try to group together and fulfil tasks co-operatively or if they compete with each other for scarce resources.

To solve this we will examine the usage of the above-mentioned filtering techniques for communication and message traffic. In conjunction with the ability to combine different views of the system, borrowing from the idea of "debugging via corroboration" introduced by Nwana et al. [8], we hope to facilitate a deeper understanding of the overall system behaviour.

For further support of debugging and steering the DIET core layer will provide functionality to influence the DIET world, e.g. by creating and killing infohabitants via direct user input. The visualisation and debugging platform will provide functionality allowing the building of an appropriate graphical user interface on top of the DIET core layer to support the development of infohabitants that directly communicate with the user.

7 Conclusions

In this paper we have presented a three-layered architecture for visualisation and debugging of an ecosystem-inspired multi-agent system, which tackles the special requirements of such a system. This architecture offers the possibility to visualise a wide range of possible applications with no changes to actual application code, good scalability with respect to the application, and in conjunction with flexible filtering capabilities to handle huge amounts of events, and a highly customisable set of views, together with the ability to easily generate new tailor-made views. This architecture shall serve as a framework for future investigations into the visualisation of adaptive and emergent behaviour of lightweight agents.

Acknowledgements. This work was carried out as part of the DIET (Decentralised Information Ecosystems Technologies) project (IST-1999-10088), within the Universal Information Ecosystems initiative of the Information Society Technology Programme of the European Union.

We thank the other participants in the DIET project, from the Intelligent Systems Laboratory, BTexact Technologies, the Departmento de Teoria de Señal y Comunicaciones, Universidad Carlos III de Madrid and the Department of Electronic and Computer Engineering, Technical University of Crete, for their comments and contributions concerning visualisation of information ecosystems. We also would like to acknowledge the support of the DFKI Intelligent Visualisation and Simulation Systems Department group members during the preparation of this paper.

References

1. Future and Emerging Technologies, European Commission: Universal Information Ecosystems web site: http://www.cordis.lu/ist/fetuie.htm, (1999).
2. R.H. Waring: Ecosystems: fluxes of matter and energy. In: Ecological Concepts. J. M. Cherrett (ed.), Blackwell Scientific, (1989).
3. DIET Project web site: http://www.dfki.uni-kl.de:8080/DIET/, (2000).
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, (1995).
5. P. Marrow, M. Koubarakis, R.H. van Lengen, F. Valverde-Albacete, et. al.: Agents in Decentralised Information Ecosystems: The DIET Approach. In: Proceedings of Symposium on Information Agents for E-commerce, AISB01 (2001 Artificial Intelligence and Simulation of Behaviour) Convention, (2001).
6. E. Kraemer, J. Stasko: The Visualisation of Parallel Systems: An Overview. In: Journal of Parallel and Distributed Computing, Vol. 18, No. 2, pp. 105-117, (1993).
7. M. H. van Liedekerke, N. M. Avouris: Debugging Multi-agent Systems. In: Information and Software Technology, Vol. 37, No. 2, pp. 102-112, (1995).
8. H. S. Nwana, D. T. Ndumu, L. C. Lee, J. C. Collins: ZEUS – A Toolkit for Building Distributed Multi-Agent Systems. In: Proceedings of PAAM98, London, UK, (1998).
9. D. T. Ndumu, H. S. Nwana, L. C. Lee, J. C. Collins: Visualising and Debugging Distributed Multi-agent Systems. In: Proceedings of Autonomous Agents 99, ACM Press, pp. 326-333, (1999).
10. Z. Cui, B. Odgers, M. Schroeder: An In-Service Agent Monitoring and Analysis System. In: Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence, IEEE Press., Chicago, USA, pp. 237-244, (1999).

Author Index