Lecture Notes in Computer Science 2789
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

László Böszörményi   Peter Schojer (Eds.)

Modular Programming Languages

Joint Modular Languages Conference, JMLC 2003
Klagenfurt, Austria, August 25-27, 2003
Proceedings

Springer

Series Editors

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

Volume Editors

László Böszörményi
Peter Schojer
Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik
Universitätsstraße 65-67, 9020 Klagenfurt, Austria
E-mail:{laszlo.boeszoermenyi,pschojer}@itec.uni-klu.ac.at

Cataloging-in-Publication Data applied for

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.3, D.2, D.1, D.4, F.3

ISSN 0302-9743
ISBN 3-540-40796-0 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10931820 06/3142 5 4 3 2 1 0

Preface

The mission of JMLC is to explore the concepts of well-structured rogramming languages and software, and those of teaching good design and programming style. A special interest is the construction of large and distributed software systems. JMLC – in the future it may be renamed to PMLC (Programming Methodology and Languages) – has already a long tradition; earlier conferences were held in 1987 (Bled, Slovenia), 1990 (Loughborough, UK), 1994 (Ulm, Germany), 1997 (Linz, Austria) and 2000 (Zürich, Switzerland).

JMLC 2003 attracted 47 papers from 10 countries (Australia, Austria, Czech Republic, Germany, Hungary, Italy, Switzerland, Russia, UK, USA). Each paper was reviewed by three representatives of the international program committee. Seventeen papers were accepted as full and 10 further papers as short contributions. In a second, simplified reviewing process the editors of this proceedings checked that all authors had accounted for the detailed comments of the reviewers.

In addition to the regular program, JMLC 2003 invited five distinguished speakers: Niklaus Wirth (ETH Zürich), Michael Franz (UCI, Irvine), Jayadev Misra (UT, Austin), C.A.R. Hoare (Microsoft Research and Oxford University) and Jim Miller (Microsoft Corporation), the latter two in common with the co-located EuroPar 2003 conference.

JMLC 2003 invited four tutorials: On *.NET* (Hanspeter Mössenböck, Wolfgang Beer, Dietrich Birngruber and Albrecht Wöß, University Linz and TechTalk); *C#* (Judith Bishop, University of Pretoria and Nigel Horspool, University of Victoria); and *Design by Contract and the Eiffel Method* and *Trusted Components and Attempts at Proofs* (Bertrand Meyer, ETH Zürich and Eiffel Software).

A memorial panel and exhibition were organized in honor of the recently deceased great computing scientists Ole-Johan Dahl, Edsger W. Dijkstra and Kristen Nygaard. The virtual part of the exhibition has been made available for everybody over the conference Web-site (http://jmlc-itec.uni-klu.ac.at/).

I thank for their generous support the University of Klagenfurt, the Carinthian Economic Funds (KWF), the Lakeside Science and Technology Park, the City of Klagenfurt, the Ministry of Education, Science and Culture, the Ministry of Transportation, Innovation and Technology, and the companies Microsoft Research, Quant-X and Uniquare. I thank Springer-Verlag for publishing this proceedings and dpunkt.verlag for sponsoring the best paper awards. I thank the authors for their contributions and the reviewers for their excellent work which was the guarantor for the high quality of this proceedings. Last, but not least, I thank the local organization team for their enthusiastic work, especially Martina Steinbacher, Angelika Rossak, Peter Schojer, Hermann Hellwagner and Harald Kosch.

June 2003 Laszlo Böszörményi, General Chair JMLC 2003

Organization

JMLC 2003 was organized by the Institute of Information Technology, University of Klagenfurt.

JMLC Steering Committee

Böszörményi, Laszlo	University Klagenfurt, Austria
Franz, Michael	University of California, Irvine, USA
Gutknecht, Jürg	ETH Zürich, Switzerland
Mössenböck, Hanspeter	University of Linz, Austria
Szyperski, Clemens	Microsoft Research, Redmond, USA
Wirth, Niklaus	ETH Zürich, Switzerland

JMLC 2003 Local Organization

Böszörményi, Laszlo	University of Klagenfurt, Austria (Chair)
Kosch, Harald	University of Klagenfurt, Austria
Hellwagner, Hermann	University of Klagenfurt, Austria
Schojer, Peter	University of Klagenfurt, Austria
Steinbacher, Martina	University of Klagenfurt, Austria

JMLC 2003 Programme Committee

Böszörményi, Laszlo	University of Klagenfurt, Austria (Chair)
Assmann, Uwe	RISE, Sweden
Back, Ralph	Abo Akademi University, Turku, Finland
Benton, Nick	Microsoft Research Cambridge, UK
Franz, Michael	University of California, Irvine, USA
Goerigk, Wolfang	University of Kiel, Germany
Gough, John	Queensland University, Australia
Gruntz, Dominik	Fachhochschule Aargau, Switzerland
Gutknecht, Jürg	ETH Zürich, Switzerland
Hitz, Martin	University of Klagenfurt, Austria
Hoare, Tony	Oxford University, UK, and Microsoft Research, USA
Horspool, Nigel	University of Victoria, Canada
Jazayeri, Mehdi	Technical University of Vienna, Austria
Ketz, Helmut	Fachhochschule Reutlingen, Germany
Kirsch, Christoph	University of California, Berkeley, USA
Kirk, Brian	Robinson Associates, UK
Klaeren, Herbert	University of Tübingen, Germany
Knoop, Jens	University of Dortmund, Germany
Lightfoot, David	Oxford Brookes University, UK
Madsen, Ole Lehrmann	Aarhus University, Denmark
Meyer, Bertrand	ETH Zürich, Switzerland
Mittermeir, Roland	University of Klagenfurt, Austria
Mössenböck, Hanspeter	University of Linz, Austria
Muller, Pieter	ETH Zürich, Switzerland
Norrie, Moira	ETH Zürich, Switzerland
Palsberg, Jens	Purdue University, USA
Peschel-Gallée, Frank	Microsoft Research, Redmond, USA
Pomberger, Gustav	University of Linz, Austria
Pree, Wolfgang	University of Salzburg, Austria
Rivera, Gabrio	ETH Zürich, Switzerland
Rüdiger, Roland	Univ. of Appl. Sciences, Wolfenbüttel, Germany
Schordan, Markus	CASC, Lawrence Livermore Lab, USA
Schulthess, Peter	University of Ulm, Germany
Szyperski, Clemens	Microsoft Research, Redmond, USA
Tkachov, Fyodor	INR Moscow, Russia
De Villiers, Pieter	University of Stellenbosch, South Africa
Weck, Wolfgang	Oberon Microsystems, Switzerland
Wirth, Niklaus	ETH Zürich, Switzerland
Zueff, Eugene	ETH Zürich, Switzerland

JMLC 2003 Referees

Arnout, Karine	ETH Zürich, Switzerland
Assmann, Uwe	RISE, Sweden
Back Ralph	Abo Akademi University, Turku, Finland
Benton, Nick	Microsoft Research, Cambridge, UK
Böszörmenyi, Laszlo	University of Klagenfurt, Austria
Clermont, Markus	University of Klagenfurt, Austria
Franz, Michael	University of California, USA
Goerigk, Wolfang	University of Kiel, Germany
Gough, John	Queensland University, Australia
Gruntz, Dominik	Fachhochschule Aargau, Switzerland
Gutknecht, Jürg	ETH Zürich, Switzerland
Hitz, Martin	University of Klagenfurt, Austria
Hoare, Tony	Oxford University, UK & Microsoft Research, USA
Horspool, Nigel	University of Victoria, Canada
Jazayeri, Mehdi	Technical University of Vienna, Austria
Ketz, Helmut	Fachhochschule Reutlingen, Germany
Kirsch, Christoph	University of California, Berkeley, USA
Kirk, Brian	Robinson Associates, UK
Klaeren, Herbert	University of Tübingen, Germany
Knoop, Jens	University of Dortmund, Germany
Lightfoot, David	Oxford Brookes University, UK
Madsen, Ole Lehrmann	Aarhus University, Denmark
Lilius, Johan	Abo Akademi University, Finland
Meyer, Bertrand	ETH Zürich, Switzerland
Mittermeir, Roland	University of Klagenfurt, Austria
Mössenböck, Hanspeter	University of Linz, Austria
Muller, Pieter	ETH Zürich, Switzerland
Müller-Olm, Markus	Fernuniversität Hagen, Germany
Norrie, Moira	ETH Zürich, Switzerland
Palsberg, Jens	Purdue University, USA
Peschel-Gallée, Frank	Microsoft Research, Redmond, USA
Pinello, Claudio	University of California, Berkeley, USA
Pomberger, Gustav	University of Linz, Austria
Porres-Paltor, Ivan	Abo Akademi University, Turku, Finland
Pree, Wolfgang	University of Salzburg, Austria
Rivera, Gabrio	ETH Zürich, Switzerland
Rüdiger, Roland	Univ. of Appl. Sciences, Wolfenbüttel, Germany
Schoettner, Michael	University of Ulm, Germany
Schordan, Markus	CASC, Lawrence Livermore Lab, USA
Schulthess, Peter	University of Ulm, Germany
Szyperski, Clemens	Microsoft Research, Redmond, USA
Tkachov, Fyodor	INR Moscow, Russia
De Villiers, Pieter	University of Stellenbosch, South Africa
Weck, Wolfgang	Oberon Microsystems, Switzerland
Wirth, Niklaus	ETH Zürich, Switzerland
Zueff, Eugene	ETH Zürich, Switzerland

Supporting and Sponsoring Organizations

University of Klagenfurt	http://www.uni-klu.ac.at/
Kärntner Wirtschaftsförderungsfonds	http://www.kwf.at/
Lakeside Science and Technology Park	http://www.lakeside-software.com/
City of Klagenfurt	http://www.klagenfurt.at/
Bundesministerium für Bildung, Wissenschaft und Kultur	http://www.bmbwk.gv.at/
Bundesministerium für Verkehr, Innovation und Technologie	http://www.bmvit.gv.at/
Microsoft Research	http://www.microsoft.com/
Uniquare	http://www.uniquare.com/
ACM	http://www.acm.org/
Österreichische Computer Gesellschaft	http://www.ocg.at/

Table of Contents

The Essence of Programming Languages

Niklaus Wirth

Abstract. Only a few years after the invention of the first programming languages, the subject flourished and a whole flurry of languages appeared. Soon programmers had to make their choices among available languages. How were they selected; were there any criteria of selection, of language quality? What is truly essential in a programming language? In spite of the convergence to a few, wide-spread, popular languages in recent years, these questions remain relevant, and the search for a "better" language continues among programmers.

1 Looking Back

When I became acquainted with computing in 1960, the field was strictly divided into two areas of application: Scientific computing and commercial data processing. Accordingly, there existed two sorts of computers: Those with binary floating-point arithmetic for scientific computing, and those with decimal arithmetic for data processing. Also programming languages catered to one of the two subjects: Fortran and Algol for scientific applications, Cobol for data processing. The choice of language was simple and determined by the application.

There was not much "science" to be found in the area. On one side were the electronic engineers. For them computers were objects of design and construction, and electronics was their field of expertise. On the other side were the mathematicians, who considered computers as their new tools, and mathematics was their field of knowledge. The new, artificial languages did not fit into either domain; they were met with cautious interest but hardly considered to be of scientific merit. Hence, a new "container" had to be invented. It is probably fair to claim that languages constituted the origin of the new subject to be called *Computing Sciences*. There were, of course, other constituents in the new container, but languages remained a driving force and a core subject. One important subject were algorithms. In order to publish new algorithms, a standard medium for publication had to be made available. A principal goal for Algol 60 had been a publication language closely reflecting the century-old traditions of mathematical notation. The goals of readability and good style were considered important.

As a result of this attention, a flurry of activities in language design sprouted within less than a decade. Many languages were designed in a hurry, published and praised. Sobering experiences followed when difficulties of implementation emerged, not to speak of the burdens of proper documentation, preparations of teaching mate rial, and support of implementations on different computers. All these factors caused many languages to disappear as quickly as they had appeared.

Nevertheless, the number of available languages grew, and programmers were confronted with choices for their daily tool. It was usually determined by the supplier of the hardware, or by the field of application: Fortran/Algol for numeric computations, Cobol for data processing, Lisp for artificial intelligence, and plain Assembler for system development. But there remained the lack of objective criteria to assess languages. Criteria were obscure, unscientifically vague. As a result, emotional preferences and prejudices dominated discussions. They made scientific conferences lively and rather exciting.

2 Languages Today

Is the situation different today after 40 years of continuous progress? In fact, it does not seem to have changed dramatically. True, the variety of languages for specific application areas has grown, but also have the general purpose languages become fewer . There is no longer a strong dichotomy between main areas of application. The market has closed in on a few languages supported by large companies. Understandably so, because systems have been made so complicated that programmers depend on such support to a frightening degree. This also caused focus to shift from the language to its implementation. In fact, many have become unable to draw a distinction.

The distinction is further blurred by the growing importance of program "libraries". They appear to be part of a system, of the language *and* of its implementation. Similar to hardware engineering, where activities have shifted from the construction of circuits from basic principles with gates and registers to the selection of the smallest number of best fitting – and perhaps enormously complex – chips, so have in software engineering programmers shifted from programming with elementary statements expressed in a language to selecting the most appropriate library routines, irrespective of their inner complexities.

In spite of these developments, we still find many programmers dissatisfied with their language and therefore looking for a better one. Often they switch to another language not because they are convinced of the quality of the new, but rather because they are disenchanted with the old language. Still, we miss widely accepted, sensible quality criteria.

The one widely accepted criterion is that a language must facilitate programming, must support the programmer in the drive for clear and systematic design, and must contribute to the reliability, trustworthiness and efficiency of written programs. It is from this statement that criteria must be derived, and on which languages must be evaluated.

3 Language Structure

Before embarking on a search for more specific criteria, we should have a closer look at the notion of language. Languages are, after all, spoken and heard. Programming languages are not! But languages have a structure. More precisely, they impose a structure on the sentences expressed in them. The meaning of a sentence can be derived by decomposing it into its syntactic parts, and then deriving the meaning of the whole from the meanings of the individual parts. This property is also fundamental for programming notations, and its adoption led to the term *language*. The formalization of syntax was one of the great early achievements.

From this it follows that a programming language is – or should be – a notation defined with mathematical rigor. Hence, every text (program) is unambiguously decomposable (parsable) into its structural components, from whose semantics the meaning of the entire program is derived. It follows further, that in order to be practical, the number of constructs must be reasonably small, and hence the language's definition must be a short, manageable text. Fat manuals of hundreds of pages are an unmistakable symptom of poor design and failure. The syntax must be lucid and appealing to conventions in related subjects, particularly mathematics. There is absolutely no need for sophisticated structures. The successful early development of powerful parsing algorithms had the detrimental effect that designers no longer paid attention to simplifying syntax. But a language that is difficult to parse by a computer, is also difficult to parse and understand by programmers. This is an example where the automatization of a process (parsing) was of questionable value in the end.

4 Language Semantics

Whatever the chosen language, the essence of programming is abstraction. This implies that a language's constructs, its building blocks, must be accurately defined in order to avoid misunderstandings, which could easily lead to grave consequences. The problem of defining language *structure* (syntax) with mathematical rigour has been solved in the decade of 1960; the precise definition of the constructs' semantics, however, has remained an elusive topic.

The earliest efforts were directed at defining the semantics of constructs such as assignment, conditional, or iterative statements in terms of simpler operations, that is, in terms of a simpler computer. This was called the *mechanistic method*. It was quickly recognized that it led into a dead end, as the simpler mechanism would then also have to be defined in terms of yet another, even simpler one. This resembles defining the effect of a program in terms of the compiled instruction sequence for a universal computer, perhaps a Turing Machine.

Needed was a method of reasoning about programs without executing them, without having to understand an underlying mechanism, a handle for verifying properties of programs, treating them as texts subject to mathematical manipulation. This requires the presence of a set of axioms and rules of inference. The chief obstacles were the *variables*, because through assignment they can change their value arbitrarily many times. In programming languages, variables are truly variable, whereas variables in mathematics are actually constant: One may substitute for a variable any value, but this value remains constant throughout a proof. But what useful statements can one make about something whose value changes, which does not stand for a fixed entity?

One way to solve a problem is to circumnavigate it. It so happened with functional languages. The solution lay in replacing iteration, during which variables assume different values, by recursion, where each incarnation of a function carries its own instance of a given variable. The fact remains, however, that the most characteristic property of our computers is the store with individually updatable cells, mirrored in languages as variables.

The solution to the problem of coping with genuine variables in reasoning about programs were the assertion (Floyd, 1967) and the loop invariant, leading to the *axiomatic method* of definition (Hoare, 1973). It postulates that every language construct (statement) Q is defined by the relationship between a *pre-condition* P (state of computation before execution of Q) and a *post-condition* R (state of computation after execution of Q). Thus, the language defines for each kind of statement a so-called *Hoare triple* P{Q}R. (An example will be given below). This scheme was improved by replacing triples by *predicate transformers* (Dijkstra, 1974). They are appropriate for proving total correctness instead of partial correctness only (correctness, if the computation terminates).

While disregarding details and further unresolved difficulties, we emphasize that the language's definition must be stated in terms of mathematical formulae and be free of references to any mechanism for execution. *A language must be understandable by its static rules alone, without knowledge about any possible implementation.* We explain this strong requirement by three familiar examples:

1. Numbers: Operations on numbers (integers) are defined by the axioms of mathematics. Hence there is no need to specify how numbers are represented as sequences of bits. In fact, representation *must not* be specified, as this would narrow the freedoms of implementers. Such an over-specification would break the notion of the abstraction; it would reveal to the reader properties of their (possible) representation instead of the numbers themselves.

2. Data Structures: Each element of an array is selected by a unique index, and an element of a matrix by a tuple of indices. Hence, there is no need for a programmer to know how an array is allocated within a computer store (whether row- or column-wise or anything else). In fact, this knowledge would allow the programmer to perform operations not specified by the proper abstraction alone.

3. Iterative Statements: Repetitions are typically explained in terms of their implementation with jumps. But there is no need for a programmer to know about jumps, in fact he must avoid the notion of structure-breaking jumps. Instead, the properties of an iterative statement S must be defined in terms of the effect of the iterated statement Q alone.

Let us illustrate this by the well known while statement:

S = **while** b **do** Q **end**

Given a condition (predicate) P which is left invariant by (execution of) Q, expressed by

P & b {Q} P

Then the definition of the while statement consists of

P {S} P & ~b

This is all that need be known about the semantics of the while construct S. No mechanistic explanations are needed, nor are they desirable.

Seldom can abstractions be implemented without impurity. It cannot be if infinity is involved, as in the case of numbers. Then a proper implementation guarantees that any computation transgressing the rules of the abstraction is notified or terminated. If, for example, an addition yields a sum outside the representable range, i.e. causes an overflow, then the program is aborted. The same holds for accessing an array with an index outside the declared bounds. These are assumed to be exceptional cases, occurring rarely, and their observation must not slow down a normal computation.

A different case of violation of the rules of an abstraction is (was?) the *go to* statement. It allows transferring control outside an iterative or other statement, thereby breaking the defining rules governing the construct. The difference to the cases of overflow and invalid index is that here the violator is an official feature of the language, obviously of an ill-designed language.

The requirement that a language's abstractions be precisely defined, complete and without a chance to be violated, is absolutely fundamental. Otherwise the advantage of using a high-level language is vastly diminished. We recall the situation of the programmer in the early years of high-level languages: When a computation failed, the result was at best a long list of octal numbers, a *memory dump*. This listing was absolutely useless, unless the programmer knew exactly how the compiler would translate a program into code and allocate variables onto memory cells. He had to leave the safe realm of abstractions and understand the details of compiler and computer, exactly what the language had promised he could safely avoid.

5 Data Types and Name Spaces

It has justly been said that a language is characterized not only by what it allows to be expressed, but equally much by what it prohibits to express. Every construct is inherently a restriction of "free expression", or, expressed more positively, of committing mistakes. It is also a discouragement of writing obscure code that is difficult to understand.

A good language introduces redundancy, that is, rules that impose consistency, rules that can be checked. Most rules should be checkable by a mere static scan of the program text by the compiler. Far fewer should require tests at execution time. *Testing as early as possible!*

One very successful feature of modern languages is the *typing of data*. It introduces the static specification of properties of variables, constants and functions, and it lets a compiler check the appropriate application of operators to data, consistency in the use of variables. In earlier languages, all type checks could be performed by the compiler (static typing). The introduction of type hierarchies through type extensions (inheritance) in object-oriented systems, required that some of the type consistency tests have to be deferred until program execution. If the concept is properly designed, such dynamic type tests can be executed with negligible overhead.

Another important concept of modern languages is that of *name space*, of restricting the visibility of identifiers to a specific part of the program text. The purpose is that names for objects can be chosen independent of names chosen in other parts (name spaces) of a program or system. This permits the independent development of various parts of a system, and their later binding without causing inconsistencies. The concept was basically present already in Algol 60, like that of data types. The name spaces were called *scopes*, and scopes were always nested. This scoping or block structure served its purpose very well at the time, but became insufficient with the growth of systems, demanding non-hierarchical name spaces. These were to be called *modules* (in *Mesa* and *Modula-2*) and *packages* (in *Ada*) and were typically parts of a system that could be developed and compiled separately.

It is important that scope rules must be simple and intuitive, and that binding of identifiers in one module to objects in other modules must be possible without overhead at run-time. Even the effort of binding must be minimal at link and load time. The burden must rest on the compiler. *Binding as early as possible!*

Although the concepts of data type and name space are independent, their coexistence in a language poses challenging problems to the implementer. Evidently, separate compilation must include full type checking across module boundaries. A well designed concept allows this checking also to be performed by the compiler alone, eliminating all overhead at link, load, and run times. This had first been achieved in the language Mesa.

6 Style and Taste

The concepts of structuring, of data typing, and of name spaces are considered as necessarily belonging in a modern programming language. There exist of course many other features and facilities not to be missed. However, they appear to be more specific to areas of application, and enumerating any of them here would seem superfluous and beside the point. Yet, when designing a language, they and their form must be selected very carefully, because the main goal is to mold them into a harmonious whole. In general, designers should be cautious in adopting features. The danger of missing an important facility is much smaller than the temptation to admit too many, particularly in too many different forms. A helpful guideline is to omit a feature, if the concept is already covered in another form. Familiar examples are the iterative constructs in Pascal, represented by while, repeat, and for statements.

Of course a designer cannot always stick stubbornly to fixed rules, but must be ready to make concessions and agree to compromises, if he wants his language to gain acceptance. Yet, in the author's experience, most comments of early users pertain to details rather than substance, and they reflect personal preferences, prejudices and habits. The designer is well advised to consider them with restraint, as taking every one into account easily breaks the style of a design and lets it appear as patchwork. I recall that users very often criticised details of the language representation rather than substance. Object of much well-meant advice were the form of comments, Pascal's case sensitivity (the property that capital and lower case letters are considered distinct), the absence of the underline character in identifiers, and the use of capitals in reserved words. All these are ultimately a matter of style rather than technical merit, and peoples' tastes may honestly differ.

Yet, there are perhaps elements of style backed up by sensible rules that are not obvious to everyone. In general, it is wise to adopt proven conventions, if there is no pressing reason for deviating. An example is the notation, left association, and binding strength of operators in expressions. When introducing new operators, symmetric symbols should be chosen for commutative operators (e.g. +, *, =), asymmetric ones for non-commutative operators (e.g. /, <, <=). A very unfortunate case is the use of "=" to denote assignment. Here, x = y and y = x have completely different effects. An obvious solution had been presented by Algol, denoting assignment by ":=". The grotesque consequence of letting "=" denote assignment is the necessity to use a symbol for equality different from that used over centuries. It looks like a singular arrogance of young programmers to overthrow the old convention of using = to denote equality and to impose = = instead. Note that by choosing != to denote inequality, also =! ought to be admitted, because inequality is commutative.

Another recommended rule of style is that for matters of little consequence (such as punctuation) light symbols are to be used, whereas symbols of importance must be heavy and conspicuous. An example where this rule was ignored is the use of the light symbols "{" and "}" for enclosing sequences of statements that can possibly extend over many lines. For this reason, capital BEGIN and END are more suitable.

Stepping up from the lexical to the syntactic level, we should mention the useful rule advising to avoid "open-ended" recursion. A much celebrated case was Algol's (and Pascal's) conditional statement, whose syntax is described in EBNF as follows:

S = A | **if** exp **then** S | **if** exp **then** S **else** S.

Its consequence is an ambiguity of statement and language, because, for example,

if x <= y **then if** x < y **then** A1 **else** A2

can be interpreted as

if x <= y **then** (**if** x < y **then** A1 **else** A2)

or as

if x <= y **then** (**if** x < y **then** A1) **else** A2 yielding different result in the cases x=y and x>y. The generally applicable, simple remedy is to close every prefixed construct with a corresponding closing symbol. Realizing that the if clause acts as an opening bracket, a corresponding end is appended, eliminating the ambiguity:

S = A | **if** exp **then** S [**else** S] **end**.

Stylistic arguments may appear to many as irrelevant in a technical environment, because they seem to be merely a matter of taste. I oppose this view, and on the contrary claim that stylistic elements are the most visible parts of a language. They mirror the mind and spirit of the designer very directly, and they are reflected in every program written. We leave it as an exercise to check the conformity of the following expressions and statements with the rules suggested above:

```
x += y;

x == y && y == z

x & y == y & z

z = ++x * y;  z = z++;

z = x++ + ++y;

for (int i = 0; i < n; i++) sum += i;

while (i < n) {sum += i; i++;}

do {sum += i; i++;} while (i < n);

top: sum += i; i++; if (i < n) goto top;
```

After studying those pieces of text, we duly wonder whether the goal of the language's designer had been to create a clear and lucid notation or to concoct a code with magic incantations.

7 What Happened?

The benefits of abstracting from machine code had been recognized half a century ago. In this movement, the language Algol, although in many ways imperfect, appeared as a first milestone. Now new algorithms were published in a form independent of any particular computer, in a form that was readily understood by mathematically trained scientists. The *Communications of the ACM*, opened a section entitled *Algorithms,* which widely served as a forum to disseminate, discuss, and improve new algorithmic methods thanks to a machine-independent notation. A standard was established, although it was recognized that improvements and extensions to accommodate new fields of application would soon be needed.

Half a century later, we read algorithms and programs published in respectable journals, which are formulated in cryptic notations hardly superior to early assembler code. Languages have become monstrously large and complicated, and their (probably incomplete) descriptions so voluminous that nobody dares reading them except when desperately lost. Computers have become so powerful and their usage so interactive, that a hectic trial and error process of writing programs is preferred to getting entangled in a maze called manual. Programming has thereby become harder, frustrating, and intellectually unrewarding. Products have become bulky and complicated to use. Programming education has drifted from a systematic study of basic principles, from a constructive, structured discipline, to an intuitive art of memorizing innumerable details of a language and its implementation, and of circumnavigating cliffs and edges of an immensely bulky and obscure mechanism.

What has happened? At times, even substantial progress had been made both in language design, language implementation, and programming discipline. The ancestral Algol had received due attention in academic circles, languages upholding its spirit, mending its weak spots, and extending its facilities have been designed, implemented, published, and used. Large systems were built demonstrating beyond doubt that on their basis very effective systems can be obtained. It was demonstrated that systems could be built with a fraction of manpower in a fraction of time hitherto needed. There was clear evidence that such languages encouraged programmers to think in a structured way, yielding programs that were lucid, whose errors could quite easily be located and corrected, and which were readily extensible. We refer to language and system *Oberon*.

Yet, the programmers' community at large chose to ignore these developments in spite of growing criticism about bulky and unreliable software. Programmers or their managers chose to stick to languages that were common and widely in use, and that did not constrain them to certain disciplines, such as data typing, and did not prohibit them to subject pointers and indices to arithmetic. In short, they preferred to stick to familiar grounds and habits, particularly if it was possible to do so under the disguise of a coating of modern syntactic sugar. Any kind of shortcomings were presumably easy to overcome through the application of an ever growing number of suitable tools, of symbolic debuggers, profilers, version managers, configuration managers, bug report managers, and graphical wonders and wizzards. This technological culture became a self-perpetuating phenomenon. The larger the set of sophisticated tools a programmer could command and apply, the higher he rose in the respect of managers. The larger the tools, the more expensive they became, the more valuable they were believed to be, but the more programmers became addicted to them.

E.W. Dijkstra once predicted that the influence of the discipline of programming on computer technology will merely be a ripple on the surface of the sea, compared to the storm of its influence on people's ability to think logically and constructively. But it has not happened! It never will happen, because most people lack the ability of thinking in terms of a complex abstraction, and because as a consequence they display an innate aversion against formalisms. Already in the early days, the marketing experts of computer industry were aware of this, and they propagated the lie that programming is easy and can be mastered by anyone. To this end, a language was developed which would let you express programs in a highly "natural", almost colloquial form without terse formulae. Unfortunately, abstract thinking is neglected by modern schools, too. The subsequent adaptation of programming methods, tools and languages was the inevitable consequence. Not a solid foundation and discipline was called for, but “fun with computers”. Systematic development was gradually replaced by diligent experimentation. Vastly increased computing power combined with interactive software made it all palatable. Now every programming problem can be tackled by anyone; for some it merely takes longer.

I vividly remember my frustrating experiences in 1976 in learning to program in *Smalltalk*, which supposedly was intended to make programming a child’s play. Accustomed to learn to understand a feature before applying it, and in want of a suitable manual, I frequently resorted to ask directly Smalltalk’s designers, just a few office doors away. Invariably their explanations came in terms of diagrams with boxes, stacks and pointers, raising more questions than answers. They explained their implementation, unwilling to distinguish between the language and its interpreting mechanism. Their abstractions could only be understood through the reactions of a computer! It may well be that children chiefly learn by experimentation and observation. Scientists, however, should be beyond this stage, and they must not be denied the much more powerful means of abstraction and logical deduction.

Regrettably, many computing scientists have moved away from programming practice, whereas practicing programmers have drifted away from a scientific approach. The result is manifest in a subject called *theoretical computer science*. Over the years it has gained a momentum of its own, quite disconnected from the subject of actual programming. Its protagonists teach about edifices impenetrable to programmers in the field, who in their turn work with languages quite inappropriate for application of the theorists’ ideas. Whereas, as claimed above, not everyone owns the ability to program professionally, not everyone who programs should have to understand, say, second order predicate calculus, category theory, or lattice theory in Banach spaces. Although there is a definite need for a sound theoretical basis, there is none whatsoever for theories for their own sake.

8 What Has Not Happened?

What really had *not* happened was that programmers and their educators had recognized the very essence of programming languages. They saw them as coding schemes by which to feed instructions more efficiently into computers, rather than as notations for designing new, abstract artefacts. They did not truly think in terms of the theory presented by the language. Instead, they remained tied to the computer and gradually learned how the compiler would translate data structures and statements into a machine representation. They were not willing to leave their traditional world of thoughts, and therefore were barred from reaping the fruits of powerful abstraction.

The academic community, on the other hand, has failed to realize a simple fact: Good engineers carry a responsibility and a liability for their products. Therefore, they want to understand what they build. This implies that the abstractions they employ be not too distant from the underlying machinery. At least an intuitive feeling for the appropriateness of their solutions must be present. In other words, the abstraction, the model of computation used, should reflect the real computer, while hiding particular details.

9 Conclusions

My conclusion is that as long as the power of programming, designing, thinking on a high level of abstraction, of programming with an adequate notation, and the necessity of ignoring the underlying computer, of discarding tricks and loopholes breaking rules governing the abstraction, are not properly recognised, we will not be able to significantly advance toward the goal of better software. There is no point in waiting for better strategies, better tools, better languages. They have all been available for some time, say, ten years, but the world at large has not been ready to take advantage. This will change only years after educators have grasped *the essence of programming languages*.

Languages are typically judged by the features which they present. Their choice is important and characterises the language. But the essence lies on a different level, namely in the manner the language is understood. It must be understood as an appropriate abstraction defined in completeness and without reference to underlying mechanisms and implementations. It must be understood as a means to express designs and ideas, to be read and studied by fellow humans, and not merely as a code to be fed to machinery. This is particularly important in education, where a language influences and shapes the manner of thinking, where consistency and precision of concepts, and where elegance and beauty of appearance are (or shold be) mandatory.

Before closing, let me mention another essential ingredient, one that hardly ever gets mentioned: It must be a pleasure and a joy to work with a language, at least for the orderly mind. The language is the primary, daily tool. If the programmer cannot love his tool, he cannot love his work, and he cannot identify himself with it.

Further Reading

E.W. Dijkstra. Some meditations on Advanced Programming. *Proc. IFIP Congress*, Munich, 1962.

R.S. Barton. A critical review on the state of the programming art. *Proc. Spring Joint Comp. Conf.*, 1963.

C.A.R. Hoare. Hints on Programming Language Design. Tech. Rep. Stanford Univ. Dec. 1973.

C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica, 2* (1973) 335-355

N. Wirth. On the design of Programming Languages. In *Information Processing 74*. North-Holland, 1974.

N. Wirth. Programming Languages: What to demand and how to assess them. Proc. BCS Symp. on Software Engineering, Belfast, Apr 1976.

N. Wirth and J. Gutknecht. *Project Oberon*. Addison-Wesley, 1992. ISBN 0-201-54428-8.

Safe Code—It's Not Just for Applets Anymore

Michael Franz

University of California, Irvine

Abstract. Despite numerous "trusted systems" initiatives, current systems software continues to be riddled with errors. The recent "Slammer Worm" incident shows that an adversary exploiting just one such error (a "buffer overrun") can cause tens of thousands of hosts to fail around the world in just minutes. Even more suprisingly, for this particular vulnerability there had been a patch available more than six months earlier, and yet even many of the OS manufacturer's (Microsoft's) own computers succumbed to the attack.

Two facts are becoming increasingly evident: First, operating systems (and increasingly also application programs) are becoming so large and evolving so quickly that it is getting prohibitively expensive to manually inspect every line of code for the absence of errors. Second, the current approach of "patching" errors as they are discovered is failing us—on one hand, we now have attacks that can spread worldwide in minutes. On the other hand, if not even Microsoft can keep its own computers current, then how can one expect that other organizations will apply all patches immediately as they are released, and in the correct order?

While the situation looks dire in the short term, there is encouraging news for the longer term: Recent research results from the mobile-code community indicate that the underlying problem can be solved in a fundamental manner, using type-safe programming languages in conjunction with code verification techniques. In many mobile-code systems, incoming programs from untrusted hosts are *verified* prior to execution. Verification means that the code *itself* is automatically examined before it is executed—this is very different from (cryptographic) *code signing*, in which only the transport envelope of the code is checked and the code is trusted blindly if the check succeeds.

We envision a future in which even the operating system is verified *prior to every execution* and have begun implementing such a solution as a proof of concept. The only thing that needs to be trusted in our architecture is a minimal safe-code platform *core* (encompassing a verifier and a small dynamic code generator), small enough to be manually audited using techniques appropriate for mission-critical software, such as fly-by-wire control systems. The core is sealed along with the processing unit into a tamper-proof hardware implement. Everything above this layer is verified, i.e., even the code in the root directory of the local hard drive need no longer be trusted.

1 Background

In January 2003, the fastest-spreading self-propagating Internet attack to date infected nearly 90% of all vulnerable hosts worldwide in under *ten minutes*. The tiny program at the heart of the attack (later called the "Slammer worm") consisted of just 376 bytes of code, yet it caused crippling slowdowns, brought down vital services (airline reservation systems, automatic teller machines), and rendered the Internet virtually unusable. What finally slowed its spread were not active counter-measures taken against it, but the network outages it created in the first place.

In the aftermath, a thorough analysis [MPS$^+$03] revealed the following astonishing facts:

– The number of infected hosts doubled almost every *eight seconds*.
– The attack exploited a buffer overflow vulnerability in Microsoft's SQL Server database product. This particular vulnerability had been made public six months prior to the attack, and a patch had been released even earlier.
– A number of Microsoft's own servers were affected by the attack.

Devastating as it already was, the attack could have been far worse. This particular worm carried no malicious "payload", i.e., its only purpose was propagating itself. One can only imagine what the damage would have been had it also deleted files, leaked database contents, or any of the other capabilities that had been well within its reach.

What lessons can be learned from this incident? First, the current practice of "patching" vulnerabilities as they are discovered clearly does not work, as the example of Microsoft's own infected servers shows. If the world's largest software manufacturer cannot even keep the patching regime for its own products on its own servers, then how can one expect compliance in any other large organizations? Second, the blinding speed with which the worm spread rendered real-time human intervention ineffective. By the time anyone could even think about a "patch", the worm had long already run its course.

"Slammer" was not a first of kind. In mid-2001, the "Code Red" (or "Nimda") worm utilized very similar techniques. It also exploited a buffer-overflow vulnerability, this time in Microsoft's IIS web server. However, it had a much slower infection rate, doubling the number of infected hosts only about every 37 minutes. The prototype for this kind of attack was the worm that Robert Morris, Jr. released on the Internet on November 2, 1988, exploiting a buffer-overflow in the Unix `fingerd` daemon. It affected between 5%-10% of all hosts on the Internet at that time.

The common denominator in all these attacks (and numerous others) lies in the exploitation of an unsafe programming language. In all three above cases, the language was C, and the exploited vulnerability was its inability to range-check array accesses. The CERT/CC Advisories for 2001 [cer01] enumerates a number of vulnerabilities in essential communications-, data storage-, and system administration software. This list includes software from Microsoft, Oracle, IBM, Hewlett-Packard, Cisco, as well as open source code. Most of the items in this list are buffer overflow and format string vulnerabilities. These vulnerabilities only exist because of the lack of safety in the C programming language. A survey of various well-known error tracking resources publicly available on the Internet shows that almost *half* of all exploited vulnerabilities in computers stem from buffer overruns [WFBA00].

The actual situation is even more ominous than it initially appears since buffer overflows are only the most primitive and obvious attacks on systems written in a weakly typed unsafe language such as C. They are also perhaps the easiest to detect using static analysis. There could easily be a multitude of more sophisticated attacks that exploit weak typing loopholes *other* than buffer overruns. As things stand, it is impossible to determine how many of these vulnerabilities are bona fide programming errors and how many are intentional “back doors” placed by agents of foreign nation-states working for domestic software companies. It should be assumed that several foreign intelligence services perhaps have partial “maps” of exploitable vulnerabilities in software systems. For such a foreign service, nothing worse can happen than a “hacker” accidently stumbling across such a carefully placed vulnerability, destroying its strategic value.

2 Toward a Solution: Insights from the “Mobile Code” Domain

A number of techniques have been proposed to counter the lack of safety in common infrastructure software written in C [CPM$^+$98]. Techniques that insert run-time checks [ABS94], static analyses [WFBA00] and combinations of both [NMW02] have been applied to the problem of checking pointer safety in existing C programs. These efforts have uncovered countless errors in common C programs such as the Linux Net Tools and SPECInt benchmarks—including errors that had earlier been overlooked in specific hand audits by humans searching for exactly these safety holes.

While the above techniques are fairly effective in the short-term, the underlying philosophy in most of the software systems communities (operating systems, control software, etc.) is still one of “adding patches when an error is discovered”, rather than trying to solve the basic underlying problem—the lack of type-safety.

Current commercial efforts at providing a more trustworthy computing base, such as Intel’s *LaGrande* and Microsoft’s *Next Generation Secure Computing Base* (also called *Palladium*), will not fundamentally improve the situation. These solutions are based on hardware-assisted *code signing* using cryptographic methods. While they prevent the execution of malicious code from unauthorized providers (and can be used to extort licensing fees from application developers wishing to become “authorized”), these trusted computing platforms are just as vulnerable to exploitation of programming errors as current operating systems are. And hence, we will inevitably see the equivalent of the “Slammer Worm” happening even under these systems.

Conversely, there exists a domain in which the problem has been addressed more fundamentally, namely the “mobile code” context. A considerable amount of effort has recently been invested into mobile code research. An important focus of this research has been the *safety of code supplied by an untrusted party*. Unlike cryptographic envelope techniques such as code signing, many mobile-code systems attempt to *verify* all mobile code prior to execution. Verification means determining the code’s safety *by examining the code itself* rather than where it came from. In the following, we apply these ideas much more broadly than just for “mobile code”.

Over the past few years, three main approaches have emerged for establishing the safety of code supplied by an untrusted (and potentially malicious) third party. They are, in turn, *virtual machines with code verification* [LY99], *proof-carrying code* [Nec97], and *inherently safe code formats* [ADvRF01,HSF02,ADF$^+$01].

– In virtual machines with code verification, the code is examined to ensure that the semantic gap between the source language and the virtual machine instruction format is not exploited. For example, virtual machines have general *GOTO* instructions but not all possible control flows are actually legal. As another example, the language definition of Java requires every variable to be initialized before its first use. Unless control flow is strictly linear, this property cannot be inferred trivially from the virtual machine program but requires the verifier to perform dataflow analysis.

– In proof-carrying code solutions, the code producer attaches a *safety proof* to an executable. Upon receiving the code, the recipient examines it and calculates a *verification condition* from the code. The verification condition relates to all the potentially unsafe constructs that actually occur in the executable. It is the task of the code producer to supply a proof that discharges this verification condition, or else the code will not be executed.

– In the third approach, an *inherently safe code format* is used to transport the mobile program, making most aspects of program safety a well-formedness criterion of the mobile code itself. Checking the well-formedness of such a format is much simpler than verifying bytecode. However, it requires a much more memory-intensive machinery at the code recipient's site, since inherently safe formats are based on compression using syntax and static program semantics. As a consequence, this approach is less well suited for resource-constrained client environments.

Clearly, these approaches are one step ahead of the current safety practices in general software systems. They are becoming increasingly important since *all* code is fast becoming "mobile", with program patches and whole applications increasingly distributed via the Internet. However, mobile-code techniques as they currently exist are not [yet] suited to support large applications. More fundamentally, all of the existing mobile-code solutions tacitly assume that they execute on top of a trusted and correct operating system (Fig. 1a). Our research is aimed at removing these two drawbacks.

3 FSSA: A Foundational Safe System Architecture

Any non-trivial system today consists of several layers, starting with the hardware on the very bottom, and typically an operating system right on top of that. The trouble with layered systems is that, similar to buildings, one cannot build something stable on top of a rotten foundation. In particular, one cannot build a secure system on top of an insecure or faulty operating system.

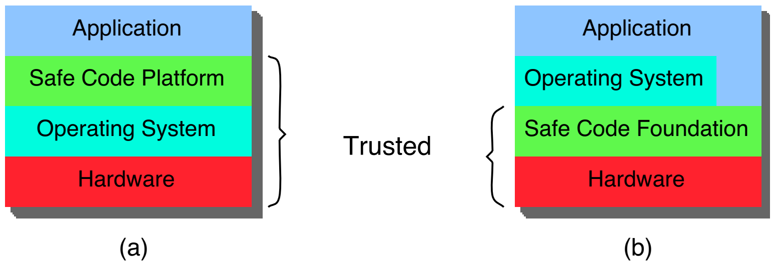


Fig. 1. (a): Typical architecture of a modern system supporting "mobile code". Note that the operating system is trusted, along with the safe code platform itself. (b): Foundational Safe System Architecture. Here, the operating system is no longer part of the trusted computing base

The problem with today's operating systems of course is that on the one hand they are very large, making it difficult to perform meaningful human audits, and on the other hand they evolve rapidly or even dynamically (in terms of downloading and opening dynamic link libraries while the system keeps running), making it difficult to use static code inspection tools.

In practice, trust in a particular implementation of an operating system is established with time, as more and more people are able to use it without major problems and more and more errors are discovered and weeded out. There is no concept of mechanically being able to check the system for compliance with a security policy. Hence, it is not possible to make any security claims about large operating systems other than "it's been out there for two decades and it seems to work". This problem becomes apparent when automated tools such as Engler's meta-compilation engine [ECCH00] are able to find hundreds of errors in operating systems that have been in wide use for a long time and were believed to be relatively correct.

But at the same time, we are now adding functionality to existing operating systems at such a pace that no part of any current mainstream operating system is really old enough to have reached the level of maturity at which one can assume that it is really "mostly correct". Add to this the new threat of system-level programmers in the employ of foreign nation-states that might be planting new backdoors in code that was thought to be mostly stable.

More recently, there have been promising attempts to shore up existing operating systems, making them less susceptible to errors. The most notable of these efforts are "Trusted BSD" [BSD], "Trusted Debian" [Deb], and NSA's Security-Enhanced Linux [NSA]. These have been able to remove some, but not all the shortcomings from their respective operating system ancestors. However, while useful and notable, these efforts will not be able to mend the more fundamental dual problems of OS size and evolution speed.

We believe the long-term solution to all these problems can only lie in *no longer trusting the operating system*. Given our deep involvement in mobile-code research, we are confident in claiming that a system can be built in which all of the operating system can be verified mechanically before execution, much in the same way that a mobile program is verified in today's mobile code systems. In such a system (Fig. 1b.) there would only be a minimal and stable piece of code (the "Safe Code Foundation") on top of the processor that would need to be trusted, small enough to be audited by humans to the highest standards. We call this approach "foundational" because safety is engineered into the system at the foundation.

4 How to Avoid Having to Trust the Operating System

We are currently working on a comprehensive security architecture that uses language-based mechanisms to eliminate errors due to circumvention of type safety, be they intentional or erroneous, and that additionally uses security policy mechanisms to contain malicious behavior. Our approach extends techniques previously applied to mobile code and is based on a combination of a) mechanically verifying the absence of such errors in any software before it is run, and b) monitoring executing software for malicious activity.

Note that language-based type safety is not the same as complete absence of run-time errors. However, run-time errors in a type safe environment do not cause spurious behaviour and cannot compromise the integrity of a system. Detection of a type error may cause abrupt program termination, but cannot be exploited for any other purpose. Also, the cause of the error is traceable in terms of constructs of the programming language—for instance "division by zero"—as opposed to details of the implementation, such as "bus error".

The main innovation of mobile-code systems, popularized by Java, is a concept of code security based on *examining the code itself.* Previous concepts of security had been based on access control (physical access, accounts, file ownership, etc.) and on cryptographic authentication (code signatures, etc.). The concept of examining the code itself to determine if it is safe gives rise to a new class of defenses against adversaries, complementing the other two (access control and authentication). The general public has largely misunderstood this, thinking that mobile code needs verification because it has an inherent defect. Nothing could be further from the truth—mobile code is showing the way to making code safer *in general*, by providing this third pillar of security in addition to the other two. For example, one could sign Java programs with a cryptographic signature, if one so desired.

Our FSSA approach takes this idea of verification from Java and similar mobile-code platforms, and puts the whole operating system on top of such a type-based safe-code platform. However, this is where the similarity ends. One cannot merely use, for example, the existing Java Virtual Machine (JVM) and simply implement the operating system on top of it. First, this would make the operating system far too inefficient. Second, the virtual machine would also be fairly large, leading back to the same problems of scale that plague today's operating systems.

Instead, our solution takes an equal part of inspiration from recent advances in Proof Carrying Code (PCC) research. Alas, existing proof-carrying code techniques have a different significant shortcoming, namely that the generated code is not portable. Once a program has been translated from source code into its executable format, the target architecture cannot be changed. Hence, using existing PCC techniques, one cannot create a long-lived and efficient architecture that can survive many generations of hardware evolution. Also, the proofs that result from this technique can be very large, much larger than the programs they relate to. One of the reasons why PCC has these very large proofs is because the level of reasoning is very low, i.e., machine code level. At this level, registers and memory are untyped, and worse still, there is no differentiation between data values and address values (pointers). A large portion of each proof typically re-establishes typing of memory locations, for example, distinguishing Integers from Booleans and from pointers.

Our approach creates a new *hybrid solution between virtual machines and proof-carrying code*, one that makes it practical to build whole operating systems on top. By raising the semantic level of the language that the proofs reason about, the proofs can become much smaller. Facts that previously required confirmation by way of proof now can be handled by axioms. Our goal is to define such a higher semantic level that is at once effective at supporting proof-carrying code in this manner, and that can also be translated efficiently into highly performing native code on continuously evolving hardware.

As an example of what we mean by “higher semantic level”, imagine a virtual machine that supports the concept of *tagged memory*, areas of memory that have a tag stored at an offset from the pointer that is unreachable via the regular memory access instructions. The virtual machine guarantees this property, i.e., the regular memory access instructions can access only locations that lie *within the data area* of a memory block—accesses are verified to lie within the range (beginning of block, end of block), which doesn’t include the tag. Conversely, access to the tag area requires one of two *privileged instructions*, only one of which reads and the other of which writes the tag value. Such an architecture greatly simplifies certain proofs: any fragment of code that doesn’t use the privileged “write-to-tag” instruction cannot have changed the tag. Since at the higher level the tag relates to the (dynamic) type of a memory object, that implies that the type has remained constant. Effectively, this design allows us to move user-level dynamic typing and garbage collection outside of the trusted code base while maintaining efficient support of memory safety.

Conversely, traditional virtual machines need to support an unlimited number of user-defined types directly at the VM level—this implies exposing to the virtual machine not only the concept of a memory tag, but also its complete semantics. As a consequence, the garbage collector needs to become part of the trusted computing base. In our architecture, the virtual machine has only a *finite* set of data types, but supplies sufficiently powerful “hooks” for efficiently supporting an unlimited number of such types at the next higher level.

Our ultimate aim is to create a safe code platform at which proof carrying code and dynamic translation can meet effectively. Hence, we also need to demonstrate the second half of the equation: how to make such a platform efficiently implementable. The key here is our use of type separation and referentially-safe encodings on one hand (as previously demonstrated in the DARPA-sponsored safeTSA project [ADvRF01]), and of an intricate memory addressing scheme on the other hand.

Hence, a central element of our architecture is a division of concerns between the proof-carrying code mechanism and the virtual machine layer: The virtual machine layer is designed in a way to reduce the burden of proof by providing a number of inherently safe operations [FCG$^+$03]. As such safe operations often involve a runtime overhead, the safe vs. the non-safe properties of the VM have to be carefully balanced. This results in including only those safe-by-construction mechanisms that have no or very little overhead compared to equivalent non-safe operations. The proof-carrying code mechanism only has to provide proof for the safety of the remaining parts of the VM architecture (Fig. 2).

5 Making It Long-Term Stable

Over the past 20 years, microprocessors have undergone dramatic changes. While 20 years might appear to be a long time in terms of academic research, industrial and military applications are often designed for a lifetime extending well beyond 20 years [Age02]. To counteract obsolescense and to reduce maintenance costs of such systems, it is essential to provide a notion of stability for the executable code format. Binary compatiblity at the machine code level is not necessarily the best solution in such a

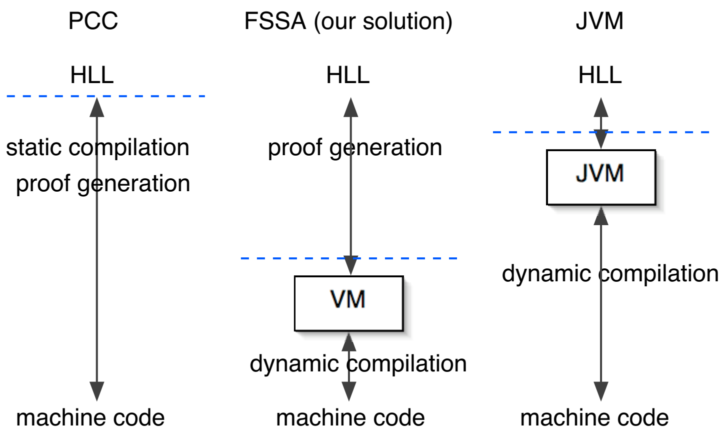


Fig. 2. Everything above the dashed line is machine independent. Compared to current PCC implementations, our framework requires shorter proofs and is machine independent. Compared to current virtual machines, dynamic compilation is simpler because the semantic distance to actual target machines is smaller. Also, we are able to perform more optimizations ahead of time

setting, because backward compatibility eventually leads either to a slow emulation of an old architecture on a new one (be it in hardware or software), or it necessitates binary translation—witness the large amounts of existing 8 bit and 16 bit code that is now being migrated to 32 bit processors. Obviously in this situation, both the emulator and/or the binary translator then become part of the trusted code base, potentially undermining security.

Hence, one of the key properties of our design is to hide the concrete target architecture by providing an abstract target architecture in form of a virtual machine. This virtual machine needs to be chosen so that it can provide efficient translation to several generations of actual instruction set architectures.

While existing virtual machines have predominantly used stack-based intermediate representations (Java [LY99], CLR [MG01]), our architecture uses typed registers for scalar values, reconciling efficiency and type-safety. In contrast to heavyweight VM approaches, our Virtual Machine has a finite set of types and does not directly deal with complex, composed types. However, to simplify the proof generation process, the memory access primitives make certain memory safety guarantees, on top of which type safety proofs can be easily built.

6 Comparison with Existing Approaches

The *language-based* approach to security [Koz99,SMH01] leverages program analysis and program rewriting to enforce security policies. Recent and promising examples of this approach include proof-carrying code [Nec97,NL98] (mentioned above), typed assembly language ("TAL") [MWCG99,MCGW98], inlined execution monitors [ES99, Sch00], and information flow type systems [Mye99].

These techniques fall into two major categories, namely program rewriting and program analysis. Program analysis covers a variety of techniques that statically try to check a program's conformance to a security policy. The primary examples of this are type-safe programming and type-based approaches to security such as typed assembly language [MWCG99,MCGW98]. Program rewriting is a complementary set of techniques that aim to enforce security by rewriting programs to conform to a security policy. Inlining security monitors [ES99] is an example of this class of techniques. An *execution monitor* (EM) monitors the execution of a *target system* and halts that system whenever it is about to violate some security policy of concern [ES99]. EMs include security kernels, reference monitors, firewalls, and most other operating system and hardware-based enforcement.

Schneider [Sch00] gives a formal characterization of the class of security policies enforceable by execution monitoring, *EM-enforceable policies*. For example, access control is EM-enforceable, while information flow and availability are not (in general).

As pointed out in [SMH01], each approach has its advantages and disadvantages, and a comprehensive, flexible, expressive and powerful security architecture would need to combine all three elements in a thoughtful manner. The primary advantage of the language-based approach to security is that it is flexible and can easily express fine-grained security policies. This is in stark contrast to the inflexible, coarse grained security provided by modern operating systems.

Meta-compilation [ECCH00] is a static technique that can automatically check conformance to certain programmer-specified high-level properties such as "every lock acquired must be released", or "these operations must be done in a certain order". This is, however, not a language-level mechanism and statically checks existing programs. DeLine et al [DF01] have elevated a similar technique to the *language level* by using types. Their language, Vault, extends C with types which keep track of resource usage and other high level properties. Vault can be used to write low-level device drivers in manner which makes it possible to check high level security properties by virtue of them being at the language level and getting automatically checked by the compiler. For example, opening a network socket and making a connection on it involves a number of library calls that must be done in a certain order. Current languages have no mechanism to check if that order is adhered to. In Vault, this can be specified and checked by the compiler by using types.

7 Conclusion

It has been decades since type-safe programming concepts were first introduced by the titans of our field, Dijkstra, Hoare, and Wirth. Commercial software developers have largely ignored these insights—and we are all suffering the consequences today. But now, growing safety concerns in a globally networked universe are presenting compelling new arguments in favor of type-safe programming. These are highly likely to bring about a renaissance of the ideas that the JMLC conference series stands for.

Acknowledgement. This research is sponsored by the National Science Foundation under grants CCR-TC-0209163, by the Defense Advanced Research Projects Agency

(DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), or any other agency of the U.S. Government.

I am grateful to my students Deepak Chandra, Andreas Gal, Vivek Haldar, and Vasanth Venkatachalam for many fruitful discussions about this material.

References

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 1994. *SIGPLAN Notices* 29(6), June 1994.

[ADF$^+$01] Wolfram Amme, Niall Dalton, Peter H. Frohlich, Vivek Haldar, Peter S. Hous el, Jeffrey von Ronn, Christian H. Stork, Sergiy Zhenochin, and Michael Franz. Project transPROse : Reconciling Mobile-Code Security with Execution Efficiency. In *DARPA Information Survivability Conference and Exposition*, June 2001.

[ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on sta tic single assignment form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147. *SIGPLAN Notices,* 36(5), May 2001.

[Age02] Defense Advanced Research Project Agency. Program Composition of Embedded Systems (PCES), SOL BAA 02-25. `http://www.darpa.mil/baa/baa02-25.htm`, 2002.

[BSD] BSD. Trusted BSD Project `http://www.trustedbsd.org/`.

[cer01] CERT Advisories. `http://www.cert.org/advisories`, 2001.

[CPM$^+$98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[Deb] Debian. Trusted Debian Project `http://www.trusteddebian.org/`.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation* [PLD01], pages 59–69. *SIGPLAN Notices,* 36(5), May 2001.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 1999. ACM SIGSAC, ACM Press.

[FCG$^+$03] Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermin Reig, and Ning Wang. A portable virtual machine target for proof-carrying code. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME03), San Diego, California*, June 2003.

[HSF02] Vivek Haldar, Christian H. Stork, and Michael Franz. The Source is the Proof. In *New Security Paradigms Workshop*, Sep 2002.

[Koz99] Dexter Kozen. Language-Based Security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-Based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.

[MG01] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. `http://research.microsoft.com/~emeijer/papers/clr.pdf`, 2001.

[MPS$^+$03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The Spread of the Sapphire/Slammer Worm. `http://www.silicondefense.com/research/sapphire/`, 2003.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Mye99] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 20–22, 1999.

[Nec97] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.

[NL98] George C. Necula and Peter Lee. Safe, Untrusted Agents using Proof-Carrying Code. In G. Vigna, editor, *Safe, Untrusted Agents using Proof-Carrying Code*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, Berlin, Germany, 1998.

[NMW02] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages*, 2002.

[NSA] National Security Agency. Security Enhanced Linux `http://www.nsa.gov/selinux/`.

[PLD01] *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[SMH01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A Language-Based Approach to Security. In *Informatics*, pages 86–101, 2001.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium, San Diego, CA*, pages 3–17, 2000.

Computing with Distributed Resources

Young-ri Choi, Siddhartha Rai, Vinay Sampath Kumar, Jayadev Misra, and Harrick Vin

Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712
{yrchoi,sid,vinay,misra,vin}@cs.utexas.edu

Dedicated to the memory of
Amit Garg, 1977–2003

Abstract. The metaphor "Network is the Computer" has received much attention lately. It is easy/hard to claim such an equivalence since neither term is defined precisely. It is easy to establish an equivalence by ignoring several key aspects of the network, such as the costs of remote data access, failures of network nodes and communication links, and the security issues inherent in distributed computing. We may then view the network as a repository of data, typically stored in distributed objects, which resembles the primary (and secondary) storage of a traditional computer. The underlying *instruction set* for the network computer consists of method calls on these objects; the effect of a method call is to modify the state of the object (similar to a *store* instruction in a traditional computer) and/or return some value (similar to a *load* instruction). Now, consider executing a high-level instruction, such as $x := f(y, z)$, on the network computer. The data represented by x, y and z may be stored at different machines, as well as the function f to be applied to y and z. An implementation of this statement has (1) to determine the sites of the data y and z, and, possibly, choose among several sites if the data are replicated, (2) choose the site where the actual computation has to take place, and (3) communicate the result to all sites where x is to be stored. The implementation could be even more elaborate when x , y and z are matrices, for example, and f is the matrix multiplication operator. Then, y and z may be sent as streams (by rows or columns), the appropriate computations are carried out, possibly by multiple computers, and the result x sent as a stream of element values as soon as such a value is computed.

What should be the structure of a high-level program for a network computer? Should a user be given the illusion that all data are locally available, there is no latency in accessing data, computations are not interleaved with computations performed by other users, and that there is never any failure? This is clearly the ideal view. Unfortunately, such a view cannot be currently supported by the internet. A user may have to confront the possibility that certain pieces of data may be unavailable, because the corresponding site has failed. The user may have to realize

that certain pieces of data could be modified by other parties during a computation.
In this paper, we develop a theory and a set of notations to represent the counterpart of a function on a network computer, i.e., how $x := f(y, z)$ should be specified and computed. Since non-determinism is inherent in a network model of computation, f need not be a function. We propose a programming model which includes non-determinism as a central concept; we call f a *task*. Task calls can be nested and tasks can be called recursively. Additionally, a task can explicitly include time-outs in its specification. Tasks capture the essence of what is currently known as *web services*.
The traditional theory of transaction processing can be used in implementing tasks; specifically, in computing $x := f(y, z)$: (1) the computation of f can be regarded as atomic, and (2) eventually, either $f(y, z)$ is assigned to x, or there is no change in the state of any object.

The Verifying Compiler: A Grand Challenge for Computing Research[*]

Tony Hoare

Microsoft Research Ltd., 7 J.J. Thomson Ave, Cambridge CB3 0FB, UK
thoare@microsoft.com

Abstract. I propose a set of criteria which distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers. As an example drawn from Computer Science, I revive an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it.

1 Introduction

The primary purpose of the formulation and promulgation of a grand challenge is to contribute to the advancement of some branch of science or engineering. A grand challenge represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale. The challenge is formulated by the researchers themselves as a focus for the research that they wish to pursue in any case, and which they believe can be pursued more effectively by advance planning and coordination. Unlike other common kinds of research initiative, a grand challenge should not be triggered by hope of short-term economic, commercial, medical, military or social benefits; and its initiation should not wait for political promotion or for prior allocation of special funding. The goals of the challenge should be purely scientific goals of the advancement of skill and of knowledge. It should appeal not only to the curiosity of scientists and to the ambition of engineers; ideally it should appeal also to the imagination of the general public; thereby it may enlarge the general understanding and appreciation of science, and attract new entrants to a rewarding career in scientific research.

An opportunity for a grand challenge arises only rarely in the history of any particular branch of science. It occurs when that branch of study first reaches an adequate level of maturity to predict the long-term direction of its future progress, and to plan a project to pursue that direction on an international scale. Much of the work required to

[*] Reprinted with the kind allowance of Springer Verlag from the proceedings of ECOOP 2003, LNCS 2743

achieve the challenge may be of a routine nature. Many scientists will prefer not to be involved in the co-operation and co-ordination involved in a grand challenge. They realize that most scientific advances, and nearly all break-throughs, are accomplished by individuals or small teams, working competitively and in relative isolation. They value their privilege of pursuing bright ideas in new directions at short notice. It is for these reasons that a grand challenge should always be a minority interest among scientists; and the greater part of the research effort in any branch of science should remain free of involvement in grand challenges.

A grand challenge may involve as much as a thousand man-years of research effort, drawn from many countries and spread over ten years or more. The research skill, experience, motivation and originality that it will absorb are qualities even scarcer and more valuable than the funds that may be allocated to it. For this reason, a proposed grand challenge should be subjected to assessment by the most rigorous criteria before its general promotion and wide-spread adoption. These criteria include all those proposed by Jim Gray [1] as desirable attributes of a long-range research goal. The additional criteria that are proposed here relate to the maturity of the scientific discipline and the feasibility of the project. In the following list, the earlier criteria emphasize the significance of the goals, and the later criteria relate to the feasibility of the project, and the maturity of the state of the art.

- **Fundamental.** It arises from scientific curiosity about the foundation, the nature, and the limits of an entire scientific discipline, or a significant branch of it.
- **Astonishing.** It gives scope for engineering ambition to build something useful that was earlier thought impractical, thus turning science fiction to science fact.
- **Testable.** It has a clear measure of success or failure at the end of the project; ideally, there should be criteria to assess progress at intermediate stages too
- **Inspiring.** It has enthusiastic support from (almost) the entire research community, even those who do not participate in it, and do not benefit from it.
- **Understandable.** It is generally comprehensible, and captures the imagination of the general public, as well as the esteem of scientists in other disciplines.
- **Useful.** The understanding and knowledge gained in completion of the project bring scientific or other benefits; some of these should be attainable, even if the project as a whole fails in its primary goal.
- **Historical.** The prestigious challenges are those which were formulated long ago; without concerted effort, they would be likely to stand for many years to come.
- **International.** It has international scope, exploiting the skills and experience of the best research groups in the world. The cost and the prestige of the project is shared among many nations, and the benefits are shared among all.
- **Revolutionary.** Success of the project will lead to radical paradigm shift in scientific research or engineering practice. It offers a rare opportunity to break free from the dead hand of legacy.
- **Research-directed.** The project can be forwarded by the reasonably well understood methods of academic research. It tackles goals that will not be achieved solely by commercially motivated evolution of existing products.
- **Challenging.** It goes beyond what is known initially to be possible, and requires development of understanding, techniques and tools unknown at the start.

- **Feasible.** The reasons for previous failure to meet the challenge are well understood and there are good reasons to believe that they can now be overcome**.**
- **Incremental.** It decomposes into identified intermediate research goals, which can be shared among many separate teams over a long time-scale.
- **Co-operative.** It calls for planned co-operation among identified research teams and research communities with differing specialized skills.
- **Competitive.** It encourages and benefits from competition among individuals and teams pursuing alternative lines of enquiry; there should be clear criteria announced in advance to decide who is winning, or who has won.
- **Effective.** Its promulgation changes the attitudes and activities of research scientists and engineers.
- **Risk-managed**. The risks of failure are identified, symptoms of failure will be recognized early, and strategies for cancellation or recovery are in place.

The tradition of grand challenges is common in many branches of science. If you want to know whether a challenge qualifies for the title ‘Grand’, compare it with

–	Prove Fermat’s last theorem	(accomplished)
–	Put a man on the moon within ten years	(accomplished)
–	Cure cancer within ten years	(failed in 1970s)
–	Map the Human Genome	(accomplished)
–	Map the Human Proteome	(too difficult for now)
–	Find the Higgs boson	(under investigation)
–	Find Gravity waves	(under investigation)
–	Unify the four forces of Physics	(under investigation)
–	Hilbert’s programme for mathematical foundations	(abandoned in 1930s)

All of these challenges satisfy many of the criteria listed above in varying degrees, though no individual challenge could be expected to satisfy all the criteria. The first in the list was the oldest and in some ways the grandest challenge; but being a mathematical challenge, my suggested criteria are considerably less relevant for it.

In Computer Science, the following examples may be familiar from the past. That is the reason why they are listed here, **not as recommendations,** but just as examples

–	Prove that P is not equal to NP	(open)
–	The Turing test	(outstanding)
–	The verifying compiler	(abandoned in 1970s)
–	A championship chess program	(completed)
–	A GO program at professional standard	(too difficult)
–	Automatic translation from Russian to English	(failed in 1960s)

The first of these challenges is of the mathematical kind. It may seem to be quite easy to extend this list with new challenges. The difficult part is to find a challenge that passes the tests for maturity and feasibility. The remainder of this contribution picks just one of the challenges, and subjects it to detailed evaluation according to the seventeen criteria.

2 The Verifying Compiler: Implementation and Application

A verifying compiler [2] uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations that are associated with the code of the program, often inferred automatically, and increasingly often supplied by the original programmer. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components. The only limit to its use will be set by an evaluation of the cost and benefits of accurate and complete formalization of the criterion of correctness for the software.

An important and integral part of the project proposal is to evaluate the capabilities and performance of the verifying compiler by application to a representative selection of legacy code, chiefly from open sources. This will give confidence that the engineering compromises that are necessary in such an ambitious project have not damaged its ability to deal with real programs written by real programmers. It is only after this demonstration of capability that programmers working on new projects will gain the confidence to exploit verification technology in new projects.

Note that **the verifying compiler itself does not itself have to be verified.** It is adequate to rely on the normal engineering judgment that errors in a user program are unlikely to be compensated by errors in the compiler. Verification of a verifying compiler is a specialized task, forming a suitable topic for a separate grand challenge.

This proposed grand challenge is now evaluated under the seventeen headings listed in the introduction.

Fundamental. Correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering. The limits of application of the theory need to be explored and extended. The project is self-contained within Computer Science, since it constructs a computer program to solve a problem that arises only from the very existence of computer programs.

Astonishing. Most of the general public, and even many programmers, are unaware of the possibility that computers might check the correctness of their own programs; and it does so by the same kind of logical methods that for thousands of years have conferred a high degree of credibility to mathematical theorems.

Testable. If the project is successful, a verifying compiler will be available as a standard tool in some widely used programming productivity toolset. It will have been tested in verification of structural integrity and security and other desirable properties of millions of lines of open source software, and in more substantial verification of critical parts of it. This will lead to removal of thousands of errors, risks, insecurities and anomalies in widely used code. Proofs will be subjected to check by rival proof tools. The major internal and external interfaces in the software will be documented by assertions, to make existing components safer to use and easier to reuse [3]. The benefits will extend also to the evolution and enhancement of legacy code, as well as the design and development of new code. Eventually programmers will prefer to confine their use of their programming language to those features and structured design patterns which facilitate automatic checks of correctness [4,5].

Inspiring. Program verification by proof is an absolute scientific ideal, like purity of materials in chemistry or accuracy of measurement in mechanics. These ideals are pursued for their own sake, in the controlled environment of the research laboratory. The practicing engineer in industry has to be content to work around the impurities and inaccuracies that are found in the real world, and often considers laboratory science as unhelpful in discharging this responsibility. The value of purity and accuracy (just like correctness) are often not appreciated until after the scientist has built the tools that make them achievable.

Understandable. All computer users have been annoyed by bugs in mass market software, and will welcome their reduction or elimination. Recent well-known viruses have been widely reported in the press, and have been estimated to cost billions of dollars. Fear of cyber-terrorism is quite widespread [6,7]. Viruses can often obtain entry into a computer system by exploiting errors like buffer overflow, which could be caught quite easily by a verifying compiler [8].

Trustworthy software is now recognised by major vendors as a primary long-term goal [9]. The interest of the press and the public in the project can be maintained, whenever dangerous anomalies are detected and removed from software that is in common use.

Useful. Unreliable software is currently estimated to cost the US some sixty billion dollars [10]. A verifying compiler would be a valued component of the proposed Infrastructure for Software Testing.

A verifying compiler may help accumulate evidence that will help to assess and reduce the risks of incorporation of commercial off-the-shelf software (COTS) into safety critical systems. The project may extend the capabilities of load-time checking of mobile proof-carrying code [11]. It will provide a secure foundation for the achievement of trustworthy software.

The main long-term benefits of the verifying compiler will be realised most strongly in the development and maintenance of new code, specified, designed and tested with its aid. Perhaps we can look forward to the day when normal commercial software will be delivered with an eighty percent chance that it never needs recall or correction by service packs, etc. within the first ten years after delivery. Then the suppliers of commercial and mass-market software will have the confidence to give the normal assurances of fitness for purpose that are now required by law for most other consumer products.

Historical. The idea of using assertions to check a large routine is due to Turing [12]. The idea of the computer checking the correctness of its own programs was put forward by McCarthy [13]. The two ideas were brought together in the verifying compiler by Floyd [14]. Early attempts to implement the idea [15] were severely inhibited by the difficulty of proof support with the machines of that day. At that time, the source code of widely used software was usually kept secret. It was generally written in assembler for a proprietary computer architecture, which was often withdrawn after a short interval on the market. The ephemeral nature and limited distribution for software written by hardware manufacturers reduced motivation for a major verification effort.

Since those days, further difficulties have arisen from the complexities of modern software practice and modern programming languages [16]. Features such as concurrent programming, object orientation and inheritance, have not been designed with the care needed to facilitate program verification. However, the relevant concepts of concurrency and objects have been explored by theoreticians in the 'clean room' conditions of new experimental programming languages [17,18]. In the implementation of a verifying compiler, the results of such pure research will have to be adapted, extended and combined; they must then be implemented and tested by application on a broad scale to legacy code expressed in legacy languages.

International. The project will require collaboration among leading researchers in America, China, India, Australasia, and many countries of Europe. Some of them are mentioned in the Acknowledgements and the References.

Revolutionary. At present, the most widely accepted means of raising trust levels of software is by massive and expensive testing. Assertions are used mainly as test oracles, to detect errors as close as possible to their place of occurrence [19]. Availability of a verifying compiler will encourage programmers to formulate assertions as specifications in advance of code, in the expectation that many of them will be verifiable by automated or semi-automated mathematical techniques. Existing experience of the verified development of safety-critical code [20,21] will be transferred to commercial software for the benefit of mass-market software products.

Research-Directed. The methods of research into program verification are well established in the academic research community, though they need to be scaled up to meet the needs of modern software construction. This is unlikely to be achieved solely in industry. Commercial programming tool-sets are driven necessarily by fashionable slogans and by the politics of standardisation. Their elegant pictorial representations can have multiple semantic interpretations, available for adaptation according to the needs and preferences of the customer. The designers of the tools are constrained by compatibility with legacy practices and code, and by lack of scientific education and understanding on the part of their customers.

Challenging. Many of the analysis and verification tools essential to this project are already available, and can be applied now to legacy code [22-27]. But their use is still too laborious, and their improvement over a lengthy period will be necessary to achieve the goals of the challenge. The purpose of this grand challenge is to encourage larger groups to co-operate on the evolution of a small number of tools.

Feasible. Most of the factors which have inhibited progress on practical program verification are no longer as severe as they were.

1. Experience has been gained in specification and verification of moderately scaled systems, chiefly in the area of safety-critical and mission-critical software; but so far the proofs have been mainly manual [20,21].

2. The corpus of Open Source Software [http://sourceforge.net] is now universally available and used by millions, so justifying almost any effort expended on improvement of its quality and robustness. Although it is subject to continuous improvement, the pace of change is reasonably predictable. It is an important part of this challenge to cater for software evolution.
3. Advances in unifying theories of programming [28] suggest that many aspects of correctness of concurrent and object-oriented programs can be expressed by assertions, supplemented by automatic or machine-assisted insertion of instrumentation in the form of ghost (model) variables and assignments to them.
4. Many of the global program analyses which are needed to underpin correctness proofs for systems involving concurrency and pointer manipulation have now been developed for use in optimising compilers [29].
5. Theorem proving technology has made great strides in many directions. Model checking [30–33] is widely understood and used, particularly in hardware design. Decision procedures [34] are beginning to be applied to software. Proof search engines [35] are now well populated with libraries of application-dependent theorems and tactics. Finally, SAT checking [36] promises a step-function increase in the power of proof tools. A major remaining challenge is to find effective ways of combining this wide range of component technologies into a small number of tools, to meet the needs of program verification.
6. Program analysis tools are now available which use a variety of techniques to discover relevant invariants and abstractions [37-39]. It is hoped that that these will formalize at least the program properties relevant to its structural integrity, with a minimum of human intervention.
7. Theories relevant for the correctness of concurrency are well established [40-42]; and theories for object orientation and pointer manipulation are under development [43,44].

Incremental. The progress of the project can be assessed by the number of lines of legacy code that have been verified, and the level of annotation and verification that has been achieved. The relevant levels of annotation are: structural integrity, partial functional specification, specification of total correctness. The relevant levels of verification are: by testing, by human proof, with machine assistance, and fully automatic. Most software is now at the lowest level – structural integrity verified by massive testing. It will be interesting to record the incremental achievement of higher levels by individual modules of code, and to find out how widely the higher levels are reasonably achievable; few modules are likely to reach the highest level of full verification.

Cooperative. The work can be delegated to teams working independently on the annotation of code, on verification condition generation, and on the proof tools.

1. The existing corpus of Open Source Software can easily be parcelled out to different teams for analysis and annotation; and the assertions can be checked by massive testing in advance of availability of adequate proof tools.
2. It is now standard for a compiler to produce an abstract syntax tree from the source code, together with a data base of program properties. A compiler that exposes the syntax tree would enable many researchers to collaborate on program analysis al-

gorithms, test harnesses, test case generators, verification condition generators, and other verification and validation tools.

3. Modern proof tools permit extension by libraries of specialized theories [34]; these can be developed by many hands to meet the needs of each application. In particular, proof procedures can be developed that are specific to commonly used standard application programmer interfaces for legacy code [45].

Competitive. The main source of competition is likely to be between teams that work on different programming languages. Some laboratories may prefer to concentrate on older languages, starting with C and moving on to C++. Others may prefer to concentrate on newer languages like Java or C#.

But even teams working on the same language and on the same tool may compete in achieving higher levels of verification for larger and larger modules of code. There will be competition to find errors in legacy code, and to be the first to obtain mechanical proof of the correctness of all assertions in each module of software. The annotated libraries of open source code will be good competition material for the teams constructing and applying proof tools. The proofs themselves will be subject to confirmation or refutation by rival proof tools.

Effective. The promulgation of this challenge is intended to cause a shift in the motivations and activities of scientists and engineers in all the relevant research communities. They will be pioneers in the collaborative implementation and use of a single large experimental device, following a tradition that is well established in Astronomy and Physics but not yet in Computer science.

1. Researchers in programming theory will accept the challenge of extending proof technology for programs written in complex and uncongenial legacy languages. They will need to design program analysis algorithms to test whether actual legacy programs observe the constraints that make each theoretical proof technique valid.
2. Builders of programming tools will carry out experimental implementation of the hypotheses originated by theorists; following practice in experimental branches of science, their goal is to explore the range of application of the theory to real code.
3. Sympathetic software users will allow newly inserted assertions to be checked dynamically in production runs, even before the tools are available to verify them.
4. Empirical Computer Scientists will apply tools developed by others to the analysis and verification of representative large-scale examples of open code.
5. Compiler writers will support the proof goals by adapting and extending the program analyses currently used for optimisation of code; later they may even exploit for purposes of further optimization the additional redundant information provided with a verified program.
6. Providers of proof tools will regard the project as a fruitful source of low-level conjectures needing verification, and will evolve their algorithms and libraries of theories to meet the needs of actual legacy software and its users.
7. Teachers and students of the foundations of software engineering will be enthused to set student projects that annotate and verify a small part of a large code base, so contributing to the success of a world-wide project.

Risk-Managed. The main risks to the project arise from dissatisfaction of many academic scientists with existing legacy code and legacy languages. The low quality of existing software, and its low level of abstraction, may limit the benefit to be obtained from the annotations. Many failures of proof are not due to an error at all, but just to omission of a more or less obvious precondition. Many of the genuine errors detected may be so rare that they are not worth correcting. In other cases, preservation of an existing anomaly in legacy software may be essential to its continuing functionality. Often the details of functionality of interfaces, either with humans or with hardware devices, are not worth formalising in a total specification, because testing gives an easier but adequate assurance of serviceability.

Legacy languages add to the risks of the project. From a logical point of view, they are extremely complicated, and require sophisticated analyses to ensure that they observe the disciplines that make abstract program verification possible. Finally, one must recognize that many of the problems of present-day software use are associated with configuration and installation management, build files, etc, where techniques of program verification seem unable to contribute.

The idealistic solution to these problems is to discard legacy and start again from scratch. Ideals are (or should be) the prime motivating force for academic research, and their pursuit gives scope for many different grand challenges. One such challenge would involve design of a new programming language and compiler, especially designed to support verification; and another would involve a re-write of existing libraries and applications to the higher standards that are achievable by explicit consideration and simplification of abstract interfaces. Research on new languages and libraries is in itself desirable, and would assist and complement research based on legacy languages and software.

Finally, it must be recognized that a verifying compiler will be only part of a integrated and rational tool-set for reliable software construction and evolution, based on sound scientific principles. Much of its use may be confined to the relatively lower levels of verification. It is a common fate of grand challenges that achievement of their direct goal turns out to be less directly significant than the stimulus that its pursuit has given to the progress of science and engineering. But remember, that was the primary purpose of the whole exercise.

Acknowledgements. The content and presentation of this contribution has emerged from fruitful discussions with many colleagues. There is no implication that any member of this list actually supports the proposed challenge. Ralph Back, Andrew Black, Manfred Broy, Alan Bundy, Michael Ernst, David Evans, Chris George, Mike Gordon, Armando Haeberer, Joseph Halpern, He Jifeng, Jim Horning, Gilles Kahn, Dick Kieburtz, Butler Lampson, Rustan Leino, John McDermid, Bertrand Meyer, Jay Misra, J Moore, Oege de Moor, Greg Morrisett, Robin Milner, Peter O'Hearn, Larry Paulson, Jon Pincus, Amir Pnueli, John Reynolds, John Rushby, Natarajan Shankar, Martyn Thomas, Niklaus Wirth, Jim Woodcock, Zhou Chaochen, and many others.

This article is an extended version of a contribution [46] to the fiftieth anniversary issue of the *Journal of the ACM*, which was devoted to Grand Challenges in Computer Science. It is published here with the approval of the Editor and the kind permission of the ACM.

References

[1] J. Gray, What Next? A Dozen Information-technology Research Goals, MS-TR-50, Microsoft Research, June 1999.
[2] K.M. Leino and G. Nelson. An extended static checker for Modula-3. *Compiler Construction:, CC'98*, LNCS 1383, Springer, pp. 302–305., April 1998.
[3] B. Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997
[4] A. Hall and R. Chapman: Correctness by Construction: Developing a Commercial Secure System, IEEE Software 19(1): 18–25 (2002)
[5] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In USENIX Annual Technical Conference, Monterey, CA, June 2002.
[6] See http://www.fbi.gov/congress/congress02/nipc072402.htm, a congressional statement presented by the director of the National Infrastructure Protection Center.
[7] F.B. Schneider (ed), *Trust in Cyberspace*, Committee on Information Systems Trustworthiness, National Research Council (1999),
[8] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Network and Distributed System Security Symposium, San Diego, CA, February 2000
[9] W.H. Gates, internal communication, Microsoft Corporation, 2002
[10] Planning Report 02-3. The Economic Impacts of Inadequate Infrastructure for Software Testing, prepared by RTI for NIST, US Department of Commerce, May 2002
[11] G. Necula. Proof-carrying code. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), January 1997
[12] A.M. Turing, Checking a large routine, *Report on a Conference on High Speed Automatic Calculating machines,* Cambridge University Math. Lab. (1949) 67–69
[13] J. McCarthy, Towards a mathematical theory of computation, Proc. IFIP Cong. 1962, North Holland, (1963)
[14] R.W. Floyd, Assigning meanings to programs, *Proc. Amer. Soc. Symp. Appl. Math.* **19,** (1967) pp. 19–31
[15] J.C. King, A Program Verifier, PhD thesis, Carnegie-Mellon University (1969)
[16] B. Stroustrup, *The C++ Programming Language*, Adison-Wesley, 1985
[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ, OOPSLA`99, pp. 132–146, 1999.
[18] Haskell 98 language and libraries: the Revised Report, Journal of Functional Programming 13(1) Jan 2003.
[19] C.A.R. Hoare, Assertions, to appear, Marktoberdorf Summer School, 2002.
[20] S. Stepney, D. Cooper and J.C.P.W. Woodcock, An Electronic Purse: Specification, Refinement, and Proof, PRG-126, Oxford University Computing Laboratory, July 2000.
[21] A.J. Galloway, T.J. Cockram and J.A. McDermid, Experiences with the application of discrete formal methods to the development of engine control software, Hise York (1998)
[22] W.R. Bush, J.D. Pincus, and D.J. Sielaff, A static analyzer for finding dynamic programming errors, Software – Practice and Experience 2000 (30): pp. 775–802.
[23] D. Evans and D. Larochelle, *Improving Security Using Extensible Lightweight Static Analysis*, IEEE Software, Jan/Feb 2002.
[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler, A System and Language for Building System-Specific Static Analyses, PLDI 2002.
[25] G.C. Necula, S. McPeak, and W. Weimer, CCured: Type-safe retrotting of legacy code. In 29th ACM Symposium on Principles of Programming Languages, Portland, OR, Jan 2002
[26] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers, Proceedings of the 10th USENIX Security Symposium, 2001
[27] D. Evans. Static detection of dynamic memory errors, SIGPLAN Conference on Programming Languages Design and Implementation, 1996

[28] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*, Prentice Hall, 1998.
[29] E. Ruf, Context-sensitive alias analysis reconsidered, Sigplan Notices, 30 (6), June 1995
[30] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991
[31] A.W. Roscoe, Model-Checking CSP, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice-Hall International, pp. 353–378, 1994
[32] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, DL Dill. CMC: A pragmatic approach to model checking real code, to appear in OSDI 2002.
[33] N. Shankar, Machine-assisted verification using theorem-proving and model checking, *Mathematical Methods of Program Development,* NATO ASI Vol.138, Springer, pp. 499–528 (1997)
[34] M.J.C Gordon, HOL: A proof generating system for Higher-Order Logic, *VLSI Specification, Verification and Synthesis*, Kluwer (1988) pp. 73—128
[35] N. Shankar, PVS: Combining specification, proof checking, and model checking. FMCAD '96,LNCS 1166, Springer, pp. 257–264, Nov 1996
[36] M. Moskewicz, C .Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, 38th Design Automation Conference (DAC2001), Las Vegas, June 2001
[37] T. Ball, SK Rajamani, Automatically Validating Temporal Safety Properties of Interfaces, *SPIN 2001,* LNCS 2057, May 2001, pp. 103–122.
[38] J.W. Nimmer and M.D. Ernst, Automatic generation of program specifications, *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, 2002, pp. 232–242.
[39] C. Flanagan and K.R.M. Leino, Houdini, an annotation assistant for ESC/Java. *International Symposium of Formal Methods Europe 2001,* LNCS 2021, Springer pp. 500–517, 2001
[40] R. Milner, *Communicating and Mobile Systems: the pi Calculus*, CUP, 1999
[41] A.W. Roscoe, *Theory and Practice of Concurrency*, Prentice Hall, 1998
[42] K.M. Chandy and J. Misra, *Parallel Program Design: a Foundation*, Adison-Wesley, 1988
[43] P. O'Hearn, J. Reynolds and H. Yang, Local Reasoning about Programs that Alter Data Structures, Proceedings of CSL'01 Paris, LNCS 2142, Springer, pp. 1–19, 2001.
[44] C.A.R. Hoare and He Jifeng, A Trace Model for Pointers and Objects, ECOOP, LNCS 1628, Springer (1999), pp. 1–17
[45] A. Stepanov and Meng Lee, Standard Template Library, Hewlett Packard (1994)
[46] C.A.R. Hoare, The Verifying Compiler: a Grand Challenge for Computer Research, JACM (50) 1, pp. 63–69 (2003)

Evolving a Multi-language Object-Oriented Framework: Lessons from .NET

Jim Miller

Microsoft Corporation, USA

Abstract. In 2001 Microsoft shipped the first public version of its Common Language Runtime (CLR) and the associated object-oriented .NET Framework. This Framework was designed for use by multiple languages through adherence to a Common Language Specification (CLS). The CLR, the CLS, and the basic level of the .NET Framework are all part of International Standard ISO/IEC 23271. Over 20 programming languages have been implemented on top of the CLR, all providing access to the same .Net Framework, and over 20,000,000 copies have been downloaded since its initial release.

As a commercial software vendor, Microsoft is deeply concerned with evolving this system. Innovation is required to address new needs, new ideas, and new applications. But backwards compatibility is equally important to give existing customers the confidence that they can build on a stable base even as it evolves over time. This is a hard problem in general, it is made harder by the common use of virtual methods and public state, and harder still by a desire to make the programming model simple.

This talk will describe the architectural ramifications of combining ease-of-use with system evolution and modularity. These ramifications extend widely throughout the system infrastructure, ranging from the underlying binding mechanism of the virtual machine, through program language syntax extensions, and into the programming environment.

The KITE Application Server Architecture

Josef Templ

Software Templ OEG
Josef.Templ@aon.at
http://members.aon.at/software-templ

Abstract. An application server represents a framework for server applications, which are programs that provide an API rather than a GUI and allow remote access to their functionality. The architecture of the Java-based application server KITE results from a generalization of the single-user desktop operating environment Oberon essentially by turning the global module list into a data structure associated with a particular client. We therefore introduce the notion of a *Service* as a named set of functions with state, the notion of a *ServiceContext* as an extensible set of service instances, the notion of a *ContextFactory* as the foundation for servers and clusters and the notion of an *Application* as a set of available services. We describe how these concepts are mapped onto the programming language Java and we discuss the fundamental implementation techniques being used. Finally we compare our approach with Enterprise Java Beans.

1 Introduction

During the development of a large scale distributed e-business application called *ec2use* by the austrian corporation *Infoniqa* it became clear that the requirements could not be met with existing standard products such as Enterprise Java Beans (EJB [5] [1]) or other middleware products.

Ec2use had to be used in a variety of ways ranging from a distributed multi-tier setup with a separate database server, a separate application server and separate online web- and Java-clients down to an offline configuration on a laptop computer with no more than 256 MB main memory where all components were running locally and periodic synchronization steps were used to exchange information with a central server. The application must not be coded twice for the different configurations, it must perform well even over low bandwidth modem connections and it must guarantee privacy of the transferred data.

The application had to be platform independent and stable, which lead to choosing Java[2] as the programming language.

In order to keep resource usage manageable, it has been decided to develop the application including the middleware (based on Java RMI [1]) from scratch, which resulted in a working application but at the expense of an ad-hoc design, which lacked clear concepts, the possibility of reuse for similar applications and a mechanism for custom specific extensions. Therefore it has been decided to redesign the application and to factor out the application independent part as an *application server.* This effort finally resulted in the architecture described in this paper.

2 Developing the Architecture

We consider an application server as a program framework for server applications, which are applications that provide an application programming interface (API) for possibly remote clients rather than a user interface (UI) for local users as it is the case for traditional (desktop) applications. An application server helps to develop distributed applications in a number of ways:

– The application server defines the overall structure of a client/server application and thereby relieves the developer from many difficult design decisions.
– It introduces a server-side component architecture and allows for reuse of components.
– It provides the infrastructure for connecting a remote client to a server.
– It supports low bandwidth connections between client and server by means of compression techniques.
– It guarantees privacy and integrity of transmitted data even over untrusted networks by means of data encryption.
– It supports scalability of the server for a large number of clients.
– It provides high availability of the server and a certain amount of fault tolerance.
– It allows the crossing of fire wall boundaries between client and server.
– It supports the common case of database applications by means of connection pooling, object/relational mapping and managed persistency.
– It provides caches for both clients and servers in order to minimize network and database load.
– It provides the infrastructure for managing transactions.
– It allows the configuration of applications such that they can be used either remotely or locally without any change in the program code.

Despite the differences to traditional applications, we can derive the overall application server architecture by a careful look at a modern single user desktop operating environment, viz. the Oberon system[7], and some generalization steps.

The Oberon system contains two fundamental global data structures, the *module list* and the *task list.*

2.1 Generalizing the Module List

Oberon's module list keeps track of all modules that were either requested for execution of a command or that have been imported by another module. Every module is loaded only once and remains loaded afterwards. Oberon modules not only provide the notion of a command (as a parameterless exported procedure), but also provide an arbitrary API to be used by other (higher level) modules.

If we want to serve multiple clients, a natural approach would be to provide a module list for every client. This is in fact the base idea of the KITE architecture, but it is expressed in Java and tuned to scale well with a large number of clients.

Similar to Oberon, Java provides the concept of a class loader, which essentially plays the role of the module list. A class loader keeps track of all classes that are loaded. A class is loaded only once per class loader and it may provide an API to be used by other classes. In addition to Oberon, Java allows to create multiple class loaders. This may lead to the idea that a class loader per client could be used as the basic application server architecture. However, using a separate class loader per client would result in loading a class multiple times, which in Java would mean that the code is allocated multiple times in main memory[1]. In order to avoid this unnecessary code duplication, we introduce a data structure per client called a *service context*. This data structure acts like a class loader but it does not keep track of loaded classes but of class instances (objects) created per client. We refer to such objects as a *service*. A KITE application represents the set of available service classes much like the Oberon system provides access to all available modules.

2.2 Generalizing the Task List

Oberon's task list keeps track of all active tasks, which are procedures that are to be called by the system at some time between commands. They act as background activities. In the context of an application server it may also be required to carry out background tasks on the server. In order to keep applications separated, a task list per application is appropriate. Since Java supports multithreading as opposed to Oberon's cooperative multitasking, we can express tasks as threads and create a thread group for every application. There is no need to provide extra support for tasks per client, since those can be created under the control of services. Only global tasks that do not belong to a particular client must be supported separately.

2.3 Server and Cluster

In order to abstract from the details of creating a connection for a remote or local client, we introduce a mechanism, called a *context factory*, which returns a service context for a specified application. A KITE application server is nothing but a context factory, which maintains a set of applications as specified in an appropriate configuration file. The context factory (alias server) will be registered with a standard Java RMI registry to be accessible for remote clients.

For configurations, where client and server are located in a single Java VM, we provide a much simpler context factory, which is not based on Java RMI.

It follows naturally, that a simple form of clustering can also be achieved as yet another implementation of a context factory. A cluster context factory

[1] We will show in Sect. 3.4 that we can make use of multiple class loaders but for a different purpose.

would be configured with a number of servers and may distribute requests for new contexts in a round robin style or some other strategy between the available servers.

2.4 The Static View

Figure 1 shows the static structure of the application server KITE according to a concrete example. There are five services involved, where *Service1* to *Service3* belong to *Application1* and *Service3* to *Service5* belong to *Application2*. Note that *Service3* belongs to both applications. There are two tasks *T1* and *T2* involved, where *T1* belongs to *Application1* and *T2* belongs to both applications. Above that there are two application servers configured, which contain both applications. The two servers are identical , therefore it is meaningful to arrange a simple 2-node failover cluster on top of them, which will connect clients with *Server1* in the normal case, but switches to *Server2* if *Server1* fails.

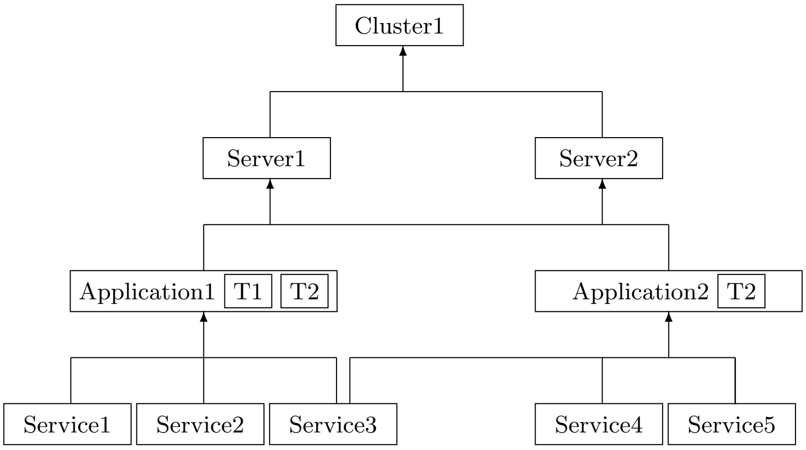


Fig. 1. The static structure of the application server KITE

The static structure closely corresponds with a hierarchy of weight as shown in Table 1. The bottom level (service) is the lightest one. A service is simply an instance of a class with the only overhead of an entry in the service context's map of services. Since there may be a large number of clients and a number of services per client, this is an important property. The next level (application) corresponds with a Java name space. It uses its own class loader, which maintains a list of loaded classes and (in current Java VMs) both the code and the static variables are replicated for every name space. An application also maintains its own profile and a thread group. The application level is considerably heavier than the service level. Tasks are somewhere between services and applications and correspond to a Java thread. The top levels (server, cluster) are the heaviest since every node corresponds to a Java virtual machine.

Table 1. Hierarchy of weight

Level	Weight	Representation
Server, Cluster	heavy	Java VM
Application	medium	Class Loader, Thread Group
Task	light	Thread
Service	flyweight	Instance

2.5 The Dynamic View

Figure 2 shows the dynamic structure of the application server KITE by applying the previous example to a particular runtime scenario. There are two clients where *Client1* uses services of *Application1* and *Client2* uses services of both *Application1* and *Application2*. The applications are represented by application objects, which may be shared by clients. On the server side, there are various services instantiated per context, but not all available services are instantiated immediately. For reasons of simplicity, the client side proxy objects for the service instances are not shown.

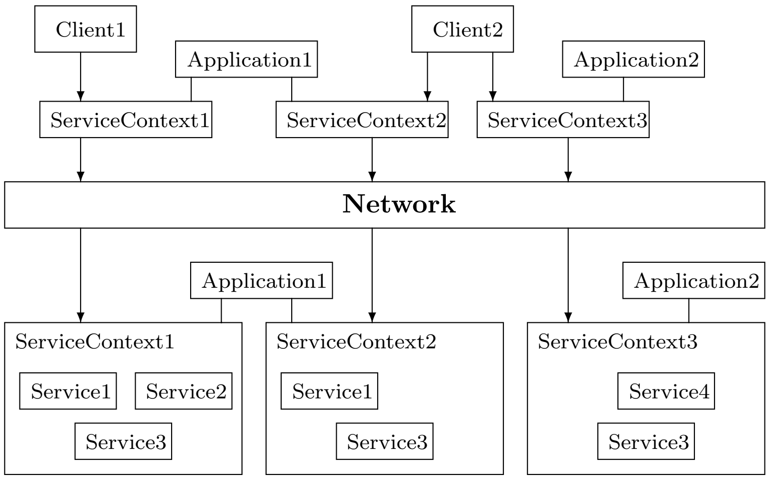


Fig. 2. The dynamic structure of the application server KITE

3 KITE Architecture API

The KITE API represents the introduced concepts as Java interfaces and classes that operate on these interfaces. In addition there are recommended naming conventions for classes and packages. The basic set of KITE interfaces and classes is contained in package *kite.service*.

We intentionally declare the base type *Exception* in the *throws*-clause of Java methods to allow implementations to throw arbitrary and yet unknown exceptions. Thereby we discourage the proliferation of exception handlers throughout the code.

For systematic exception handling KITE requests (by convention) all exceptions to be propagated to the very end of the call chain. At this place, most probably a command invocation initiated by the user of an application, there should be an exception handler, which notifies the user and resets the service context as described in Sect. 3.2.

3.1 Service

A service provides an interface and an implementation for a particular group of functions (also called *business logic*). A client may access a method on a server by means of a service. Services may refer to each other if they are within the same service context. A service is potentially stateful, i.e. it may store information (session state) across individual requests for a particular client. Technically speaking, a service is nothing but an instance of a class that implements its interface and the interface *Service*. KITE provides various standard services, which may be used by all applications. Other services may be provided by the application developer. All services follow a basic life cycle which is expressed by interface *Service*[2].

```
public interface Service {
  void init(ServiceContext ctx) throws Exception;
  void reset() throws Exception;
  void close() throws Exception;
  void setLocale(java.util.Locale locale) throws Exception;
}
```

After a service has been instantiated, it will be initialized by a call to *init(ctx)*, which passes the service context to the service. The service may perform arbitrary initialization steps including requesting other services from *ctx*. *reset()* sets the service instance into a well defined state after an exception has been caught. After reset, the service may accept further requests. For example, a database service could close open transactions upon *reset()*. *close()* marks the end of the lifecycle of a service. *setLocale()* sets the locale to be used by the service.

For an example service outline please refer to Sect. 6.

3.2 ServiceContext

A *service context* is a dynamically extensible set of instantiated and initialized services, where every service is instantiated only once. Instantiation of a service is done upon the first request for a particular service. The typical usage of a

[2] In order to minimize coding effort an auxiliary class *kite.service.ServiceAdapter* with empty method implementations has been prepared.

service context is that for a client, which connects to a server, a new service context is provided. This context holds the services the client requested together with the state of these services. A client may open an arbitrary number of service contexts to an arbitray number of servers, for example to copy data from one server to another. KITE provides implementations for service contexts for local and remote clients. In any case, service contexts are provided by KITE and need not be extended in an application dependent way.

```
public interface ServiceContext {
  Service getService(Class service) throws Exception;
  Service getService(...) throws Exception; //some variants
  void reset() throws Exception;
  void close() throws Exception;
  void setLocale(java.util.Locale locale) throws Exception;
  java.util.Locale getLocale() throws Exception;
  Application getApplication() throws Exception;
}
```

getService() returns a service instance for the requested interface. If the naming conventions of Sect. 4 are followed, the implementing class will be derived automatically.

The methods *reset()*, *close()*, and *setLocale()* will simply be forwarded to all services. *getLocale()* and *getApplication()* provide information about the currently set locale and the application, the context is associated with.

```
public interface ServerContext
       extends ServiceContext, java.rmi.Remote {
  Object[] invokeService(String serviceClass, String methodName,
    String signature, Object[] arguments) throws Exception;
  Object invokeService(String serviceClass, int methodId,
    Object[] arguments) throws Exception;
}
```

A *ServerContext* represents a specialized service context for remote access of services. It provides method *invokeService()*, which may be used to invoke a method of a service from a remote client from within its proxy (stub) object. The first variant allows to specify a method by class name, method name and signature and returns two objects, the result and a method id, which may be used in the second variant to access the method more efficiently.

3.3 ContextFactory

A *context factory* is a class which creates service contexts. KITE provides implementations for context factories for both local and remote clients and for a simple form of clustering, but in special cases (for example for a special form of clustering) an additional context factory could be implemented.

```
public interface ContextFactory {
  ServiceContext newContext(Profile profile) throws Exception;
}
```

newContext() creates a new service context. The specified profile contains the configuration parameters including the application name, version number, locale, etc.

A KITE server is simply an implementation of a context factory for remote clients, i.e. it is a remote object registered with an RMI registry and it returns an instance of a *ServerContext* with *newContext*.

3.4 Application

An *application* in the KITE framework is the set of services, which are visible via the application's class path, and a set of tasks, which are associated with the application. An application is defined by a set of configuration parameters (called *profile*) which includes name, version number, class path and profiles for the services and tasks. An application is loaded during server startup by reading the application's profile and creating a class loader for the application. The class loader is used for dynamically loading all classes and resources of the application. This results in a separation of applications in different name spaces, which minimizes interferences between applications, since all the application classes have their own code and static variables. It also allows applications to exist in multiple versions at the same time in the server. This feature is commonly known as *hot deployment*, because it allows to update an application without shutting down and restarting the server. The system classes (`java.lang`, `java.io`, `java.net`, etc.), however, are shared between the applications, which avoids the waste of resources in the server. In addition to services KITE supports the concept of Tasks, which are server side threads associated with an application. KITE provides the class *Application*, which handles the management of class loaders, tasks, etc. This class is not extended for particular applications.

3.5 Task

A *task* is a program which is associated with an application and is automatically started by the server upon loading of the application. A task always runs within the name space (class loader) of its application. Technically speaking, a task is a class that implements the interface *Task*, which extends the Java interface *Runnable*. Therefore, tasks are specialized Java threads. All tasks of an application are collected within a Java *thread group*. Tasks may use all services of the application they are associated with.

```
public interface Task extends Runnable {
  void initTask(ServiceContext ctx, kite.util.Profile taskProf);
  void stopTask();
}
```

A task is associated with a service context and a profile by means of the method *initTask()*. In order to avoid the usage of deprecated Java APIs, tasks employ the recommended pattern for stopping threads by introducing the method *stopTask()*.

4 Naming Conventions and Class Lookup

KITE organizes its classes and interfaces in a hierachy of packages. This hierarchy reflects the fact that in a client/server application some components will be needed on the server side only, others will be needed on the client side only and some may be needed on both sides. By separating classes and resources this way, it is very easy to create archives that contain only the minimum number of files for either side. The resulting package structure used and recommended is called the KITE package triplet.

4.1 The KITE Package Triplet

All classes, interfaces and resources of a component *c*, which are needed only on the server side, form a package named *c*.server. All classes, interfaces and resources of a component *c*, which are needed only on the client side, form a package named *c*.client. All classes, interfaces and resources of a component *c*, which are needed on both the client and server side, form a package named *c*.service. Applying these rules to KITE itself leads to the package triplet *kite.service*, *kite.server*, *kite.client*.

The standard services provided by KITE do not belong to the server, but to the applications loaded and managed by KITE. They must therefore be located in the application class path and not in the system class path if the advantages of hot deployment are to be used. For this reason, the KITE standard services represent a separate component available in the packages *kite.std.service*, *kite.std.server* and *kite.std.client*.

4.2 The KITE Service Triplet

For each service *S* of a component *c*, which may be called remotely, there must be a Java interface and a server side implementation class. For various reasons (e.g. caching) it may also be necessary to provide another implementation of interface *S* which will be loaded and executed on a remote client. Thus, in general there are three class files involved, where only the interface is needed on both sides. KITE uses the following naming conventions to look up an implementation class when a service is requested by specifying the interface only: *c*.service.*S*Service, *c*.server.*S*Server, *c*.client.*S*Client.

As an example, a shopping cart service of ec2use would result in the triplet consisting of *ec2use.service.CartService*, *ec2use.server.CartServer* and optionally *ec2use.client.CartClient*.

For remote clients, class lookup is performed on the client side by checking the availability of a client side implementation first. If found, this class is used, if not, a stub is created. There is no network round trip involved so far. When a method of a stub is called, this call is forwarded to the server, which checks the availability of a corresponding server side implementation. If found, a service will be instantiated and registered. This occurs as a side effect of the first remote method call of a service and minimizes the number of network round trips.

The service context for local clients only checks for the availability of a server side implementation class.

5 Implementation Aspects

The KITE communications infrastructure is based on Java RMI (Java Remote Method Invocation [1]), however, it minimizes the usage of RMI to an absolute minimum. In particular, there is an RMI interface for RemoteContextFactory, an implementation of a context factory for remote clients. This enables KITE to establish a connection between two Java VMs over the Internet. The RMI lookup returns a context factory stub object to the client. The stub allows the creation of a service context for a particular application. In the case of a remote client, the returned service context is also an RMI remote object. If the client shares the Java VM with the server, there is no RMI involved at all.

The individual services of a KITE application are (in the regular case) not RMI objects, however, they can be used locally as well as remotely. In case of remote invocations, so called *dynamic stubs* are used to mimic the semantics of RMI.

Dynamic stubs in KITE follow the idea described in [3] for Oberon but expressed in Java by means of *dynamic proxies* of package *java.lang.reflect.* A proxy class is created for every service and registered in the service context of the remote client. The proxy class implements a set of interfaces such that any method call results in activating a generic invocation handler, which simply forwards the call via method *invokeService()* of the remote service context to the server. The use of dynamic stubs results in a number of advantages:

1. No need for using the Java RMI compiler (rmic) if the interface of a service changes and less files to deploy to the client.
2. No RMI objects for services on the server. Those objects would have a significant memory overhead, which is about 600 Byte per object.
3. No additional network traffic for lookup of services, which allows exceptionally short startup times even in the case of low bandwidth connections.
4. No additional server load or network traffic through RMI's distributed garbage collector (DGC).

Java RMI uses a so called *socket factory* for creating sockets. A socket represents a bidirectional communications link between client and server. The standard RMI socket factory may be replaced by KITE's own socket factory, which results in creating sockets with extended functionality. In particular, KITE sockets allow online compression and encryption of data and the routing of requests to a servlet on an HTTP server in order to cross firewall boundaries.

6 Putting It All Together

This section outlines the resulting structure of a KITE service and compares it with a corresponding module structure in Oberon. We use a service called *MoDDLService*, which provides access to database objects in Java. MoDDLService is built upon a lower level service called *DatabaseService*.

```
MODULE MoDDLService; (* Oberon version *)
  IMPORT dbs := DatabaseService;
  ...
  PROCEDURE getRecord*(...): DbRecord;
  BEGIN ... (* use 'global' dbs *)
  END getRecord;
END MoDDLService.

public class MoDDLServer implements MoDDLService {// KITE version
  private DatabaseService dbs;
  ...
  public void init(ServiceContext ctx) { //replaces IMPORT
    dbs = (DatabaseService)ctx.getService(DatabaseService.class);
  }
  public DbRecord getRecord(...) {
    ... // use 'per client' dbs
  }
}

// KITE client; uses MoDDLService; profile name in args[0]
import ...;
public class MyClient {
  public static void main(String[] args) throws Exception {
    Profile p = new Profile(args[0]); //read configuration
    //use convenience class 'Client' for simple clients
    Client.init(p);     //initialize logging, locale, etc.
    ServiceContext ctx = Client.getContext(p);
    MoDDLService ms =
      (MoDDLService)ctx.getService(MoDDLService.class);
    DbRecord rec = ms.getRecord(...);     //use service
    ...
  }
}
```

7 Related Work

The J2EE [5] standard includes a specification of a container for so called *Enterprise Java Beans* (EJB), which addresses a similar domain as KITE does. The main differences between EJBs and KITE are summarized below.

EJBs come in multiple forms as stateless and stateful session beans, entity beans and message driven beans. KITE provides a service context, which corresponds to a stateful session bean.

EJBs requires the use of dual interfaces, one for remote and one for local clients. KITE uses the same interface for both purposes.

EJBs require the explicit specification and implementation of a so called *home interface* for every bean. KITE provides a generic context factory instead.

EJBs entity beans with remote interfaces proved to be too inefficient for practical use, because every attribute access is a remote call. KITE relies on an appropriate service to pass database objects as a whole between server and client.

EJBs neither support the concept of services nor tasks.

EJBs support distributed transactions based on the XA specification. KITE currently does not support distributed transactions. We can imagine, however, to introduce an additional service (*TransactionService*), which provides distributed transactions. The problem is, that there are only very few resource managers available that can take part in such a transaction. Most JDBC drivers, for example, are not XA enabled.

EJBs support declarative security constraints based on individual methods. KITE does not support such a feature.

Resource usage (memory, CPU, programming effort) is significantly higher for EJBs.

Acknowledgements. I would like to thank the ec2use team at Infoniqa for their support and contributions through many discussions and the Infoniqa board of directors for agreeing to publish the KITE architecture at JMLC 2003.

References

1. Flanagan, D., Farley, J. Crawford, W.: Java Enterprise in a Nutshell. O'Reilly, 1999.
2. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison Wesley, 1996, The Java Series.
3. Hof, M.: Just-in-Time Stub Generation, Proc. JMLC, Linz, 1997.
4. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 1996.
5. Sun Microsystems: Java (tm) 2 Platform, Enterprise Edition. Addison Wesley, 2000, The Java Series.
6. Templ, J.: MoDDL 1.0 language report, Infoniqa technical report, July 2001.
7. Wirth, N.: Project Oberon – The Design of an Operating System and Compiler. Addison Wesley, 1992.

Compiler Support for Reference Tracking in a Type-Safe DSM

Ralph Goeckelmann, Stefan Frenz, Michael Schoettner, and Peter Schulthess

Distributed Systems Laboratory, Ulm University o-27, D-89069 Ulm
schulthess@informatik.uni-ulm.de

Abstract. The efficiency of language implementations is heavily influenced by the selected strategy for allocation and reclaim of memory. Memory allocation in a distributed shared memory (DSM) cluster poses additional challenges. Designing the DSM as a distributed heap is natural and relieves the application programmer from the burden of memory management. Garbage collection is incremental and refrains from repeatedly marking, sweeping and writing to the distributed memory. Relocation of objects is implemented to reduce memory fragmentation and to resolve false-sharing conflicts. Reference tracking and a type-safe language are essential for garbage collection and object relocation. In this paper we present a novel data structure which we call "backpacks" used to efficiently keep track of global references in our language-based DSM. We also show how our home grown Java Compiler supports reference tracking and garbage collection by generating bi-directional runtime structures.

1 Introduction

Safe memory allocation and reclaim is based on the concept of strong typing provided by the programming language. This type safety is typically restricted to a single machine and its main memory. Once we leave the single station environment and the main memory, middleware packages take over and we lose the benefits of the safe type system. In the Plurix operating system (OS) we extend the typing introduced by the Java programming language to a cluster of workstations and provide type-safe memory-management in a distributed shared memory (=DSM) [1]. Language based OS development has been successfully demonstrated in systems like Oberon [2].

DSM systems introduced by Keedy and Li offer the abstraction of a single address space to all stations of a cluster [3, 4]. Traditional DSM environments are built on top of Unix or Microsoft Windows systems and do not exploit type-safe languages but use C with its known drawbacks. Furthermore, they offer only coarse grain memory allocation primitives and it is the burden of each application to implement allocation and de-allocation functions [5]. This approach is error-prone, time consuming, and typically causes performance penalties. We avoid these disadvantages by implementing our Java-based DSM as a global heap optimized to achieve high performance. Language objects (classes, interfaces, instances, and arrays) are simply allocated, manipulated and eventually recollected in this global heap. The latter is the responsibility of our distributed garbage collection.

Modern programming languages (e.g. Oberon, C#, Java) typically offer automatic garbage collection. As the programmer may no longer explicitly free memory, dangling references and memory leaks are avoided. In a DSM environment it is not a good idea for a distributed garbage collection (GC) scheme to mark or to colour the heap blocks during a traversal because collision probability grows with the number of written objects. The "backpack"-scheme proposed in this article is designed to support efficiently distributed garbage collection for a DSM heap avoiding these drawbacks. The "backpack"-scheme is also the foundation for a dynamic object relocation facility used to resolve false-sharing situations and to compact fragmented parts of the heap.

Our Plurix Java Compiler (PJC) is used to develop the OS and the applications. It directly translates Java source texts into Intel machine instructions and is an integral part of the OS [6]. Native code generation is mandatory for OS and driver development in Java. The PJC generates bi-directional runtime structures and supports reference tracking to simplify garbage collection and object relocation.

Our paper is composed of five sections starting with a description of bi-directional runtime structures. In section three we present the backpacks - our novel reference tracking scheme. In section four we discuss memory allocation, garbage collection and object relocation. Finally, we present selected performance measurement data.

2 Bi-directional Runtime Structures

We decided to implement bi-directional heap blocks (Fig. 1) to simplify the garbage collection process, to support the relocation of objects and to facilitate a periodic heap consistency check. A reference will always point to an object header in the middle of the block. To the left of the header we find references to other objects, including code segments and class descriptors. The first reference on the left side is always a pointer to a class descriptor defining the type of each memory block. To the right we find scalars and scalar array elements.

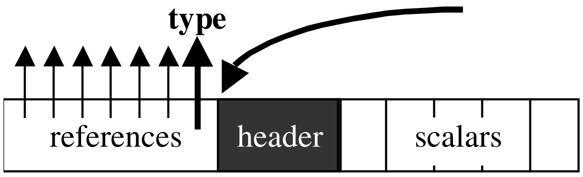


Fig. 1. Bi-directional layout scheme of heap blocks

Our Plurix Java Compiler generates all runtime structures according to this bi-directional scheme. Numerous variants of runtime structure memory layouts have been discussed for object-oriented languages especially to support multiple inheritance. To the best of our knowledge no other implementation of an object-oriented language has established a bi-directional layout for separating references from scalars. Of course bi-directional layouts are well known and have been used for other purposes.

For example Myers used it to implement multiple sub-typing in the Theta language [7]. Using a bi-directional memory layout he achieves class/super-class compatibility by separating fields from the dispatch header. The field part, however, may still contain references to other instances and differs from our approach.

Language implementations with automatic garbage collection often require voluminous offset tables to distinguish between pointer references and scalar variables whose value might look like a memory address. With bi-directional heap blocks the garbage collection only needs to scan the reference portion of the heap block and all entries are known to be references.

In principle each Java object is allocated as a separate heap-block which can reference further objects (including arrays and strings). A separate block is created even if the size of the object is known at compile time and the compiler would be in a position to embed a referenced object directly into the original object. We thus risk obtaining a heap with an unnecessary amount of fragmentation and with many small blocks which are expensive to manage and to collect.

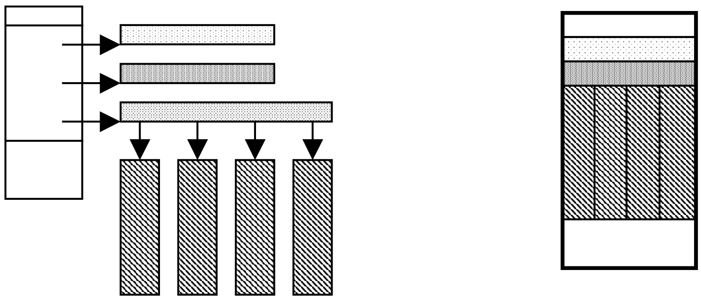


Fig. 2. Java compatible versus Plurix monolithic memory layout

This object fragmentation in the heap is particularly disturbing because the compiler must set up class descriptors. These contain scalars and arrays (e.g. for the method jump table) which in real Java would lead to complex linked list structures (Fig. 2). The compiler creates a linearized and monolithic class descriptor by bypassing the type rules. A detailed description of our runtime structures and language extensions can be found in [8].

3 Reference Tracking

To support reference tracking the Plurix Java Compiler generates a runtime call for each pointer assignment to a heap reference. This method is very short and is only used to update the bookkeeping of references. To reduce the overhead caused by the call during pointer assignment we only track heap references. Pointers residing on the stack must not be considered because garbage collection and object relocation is only executed if the stack is empty (see section 4). This is not a heavy restriction because we do not implement preemptive multithreading but transactional processing with a cooperative multitasking scheme.

3.1 Backchain – Our Old Solution

Considerable thinking effort has been spent on the run-time structures in the distributed heap of our OS prototype. An early implementation of our heap organization used a *backchain* scheme (Fig. 3) where all references to an object were linked into a serial list starting at the object itself.

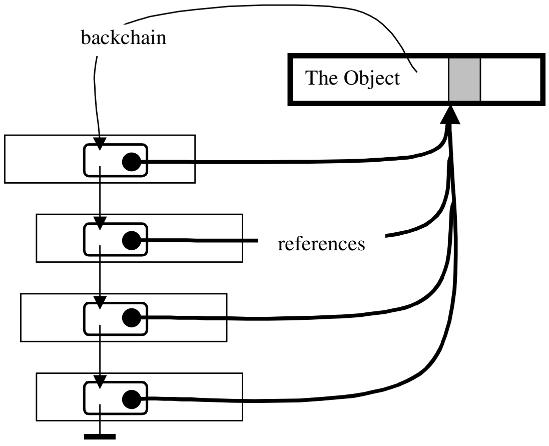


Fig. 3. Serial backchain linking all references to an object

Since each reference was 32 bits wide the full container size for a pointer was increased to 64 bits to include the backchain link. The chain conveniently supports relocation of objects and incremental garbage collection. Any object with an empty backchain is garbage and can be reclaimed. When an object is reclaimed it might itself reference some objects and thus be linked to their Backchain. It is now necessary to unlink all freed reference variables from their respective chains. This requires an expensive linear search in a chained list, and a lot of memory pages might be fetched over the network if not present locally. Furthermore, garbage cycles are not recognized and might require a separate mark and sweep procedure [9].

The backchain requires updating whenever an assignment or de-assignment to a heap reference occurs. This updating of the backchain could become costly and lead to unpredictable invalidation patterns in the distributed heap (see 4.1). New references to an object are always inserted at the head of the list and the object itself is invalidated. In case of a de-assignment any object within the backchain may be invalidated because its pointer need to be updated to reference the successor of the pointer to be detached.

Statistics showed that backlinks were mostly empty and when proceeding from a 32 bit address space to 64 bit addresses in some distant future the waste of memory will become excessive. The backchain concept was eventually discarded in favor of the backpack scheme described in the next paragraph.

3.2 Backpacks – The New Solution

Using so-called *backpacks* instead of the backchain of the early implementation proved to be superior with respect to the invalidation patterns and to the storage requirements. The scheme is illustrated in Fig. 4.

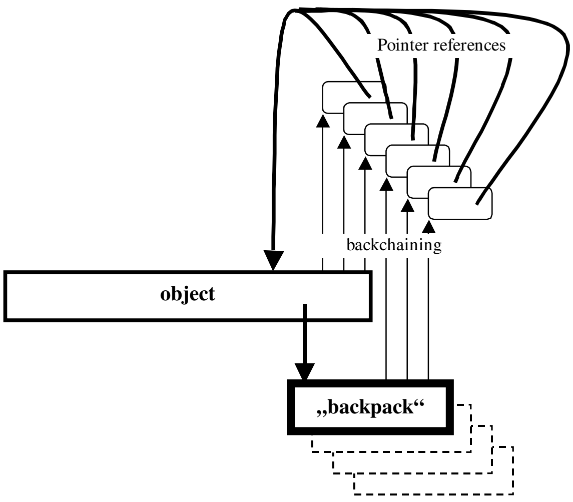


Fig. 4. Backpacks tracking the references to an object

The references to an object are tracked in a separate data structure called *backpack* which is an object by itself and contains the memory addresses of the respective pointers. The first three references are tracked from direct back-links [illegible] are located in the object itself. A backpack is thus only created if there are [illegible] than 3 global references to an object.

References to widely used systems objects, e.g. the class descriptor of "java.lang.string", are not tracked and do not trigger the creation of backpacks. In fact these objects are allocated in separate memory pool to be specified at compile time or at run time. In actuality less than 10% of all objects require the creation of a backpack, see Sect. 5. The management of backpacks adds further complexity to the memory manager but this complexity is rewarded by superior DSM performance. Updates to the bookkeeping of references require less network traffic due to the better access locality. Furthermore unpredictable invalidations caused by backchain maintenance (see 3.1) do no longer occur. Finally, the backchain approach requires double pointer size - the backpack approach instead adds overhead only when necessary. This backpack scheme has been the subject of a recent patent application.

4 Heap Management

Programming languages typically implement two separate strategies of memory allocation: stack allocation and heap allocation. Early Pascal implementations simply marked the top of the heap before proceeding with further allocation. Later all allocations beyond the mark were released in one step – thus simulating a second stack.

More sophisticated schemes are able to individually allocate and free single objects – either explicitly or by implicit automatic garbage collection. Buddy systems split and combine memory blocks of size $k*2^n$ to reduce heap fragmentation [10]. Separate free-storage chains may be maintained for different container size to reduce the search for a suitable container. The search for a chunk of memory may be "first fit", "best fit" etc. However, memory allocation in a DSM poses additional challenges.

4.1 Distributed Memory Allocation

Modifications within a node of the DSM must be propagated and coordinated. Most DSM systems implement the write-invalidate coherence protocol to invalidate all read-only copies of a memory page before a node is allowed to write it. Furthermore, memory consistency models define when write operations become visible for other stations. In a single station model any write to main memory is immediately visible for all subsequent read operations. This is known as strict consistency – the most comfortable programming model – but the most expensive in a distributed system due to excessive network latencies. As a consequence DSM designers propose weaker models delaying the propagation of write operations and thus improving memory performance [11].

Implementing a distributed heap structure within a DSM obviously requires a strict memory consistency model to avoid inconsistencies within the memory manager. Clearly it is difficult to implement strict consistency in a DSM but the Plurix project introduces *transactional consistency* solving this problem. Any command in our system is executed as a restartable transaction and concurrent TAs are serialized using an optimistic synchronization scheme [12].

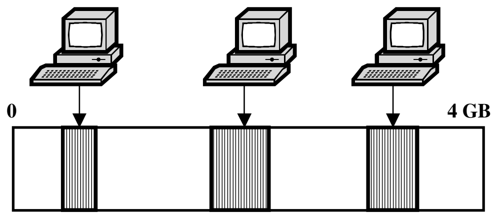


Fig. 5. Distinct heap entries per node

Memory allocation in a DSM should try to leave adjacent heap blocks unmodified to avoid potential collisions with blocks used by other nodes. Using different entry points in the heap will cluster objects allocations of each node and improves locality

(Fig. 5). Special memory pools must be used to protect vital components of the system from invalidation, e.g. kernel. More details about memory pools in [13].

4.2 Garbage Collection Using Backpacks

Reclaiming memory is the task of our distributed garbage collector (GC). The bi-directional memory layout of the runtime structures lets the GC identify pointers without additional overhead. The addresses of object references are provided by the backpacks without additional calculations. If there are no references to an object it is obviously garbage and can be collected. Using the backpacks and backlinks the GC can easily detect unreferenced objects.

Each object not reachable from the global root of the cluster-wide naming service is garbage. Detecting cyclic garbage efficiently in our DSM environment is still under study. The Plurix system saves checkpoints in short intervals on disk to survive node failures. We are considering running an off-line GC on the last DSM snapshot to identify cyclic garbage. Potential candidates to break a cycle can be searched and a hint can be posted to the running cluster.

Incremental GC is periodically executed when the station is idle (the stack is empty) or running out of virtual memory. The latter situation only occurs if the total cluster runs out of memory and will virtually disappear with the advent of 64 bit processors. Executing the GC only on an empty stack allows the reference tracking to ignore assignments to pointers residing in the stack. This is not a critical restriction as we have no multithreading per node but a transactional processing. However, multiple nodes are allowed to concurrently execute incremental garbage collectors.

Collisions between a GC and an overlapping application transaction executing on another node are detected and the GC will then abort tacitly. The object(s) that caused the collision were inspected by the GC but obviously are still in use. We plan to use such collision information as an input for the next restart of the GC.

4.3 Object Relocation

Relocation of objects is simplified because moving an object by a certain offset amount can be compensated by adjusting all references in the backpack. The transactional processing allows the object relocator to be executed concurrently with other transactions on other nodes - any collisions are detected and resolved.

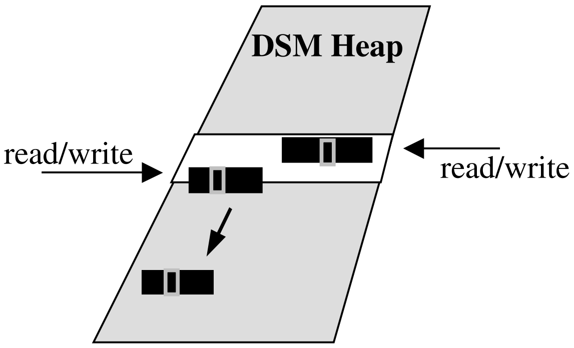


Fig. 6. Relocation of an object false-sharing a single memory page

Relocation may be necessary to reduce heap fragmentation or to control false-sharing situations. False-sharing causes significant performance penalties in page-based DSM systems because of heavy page trashing [14]. If two objects reside on a memory page and within a certain time interval each of the object is accessed by a different node access conflicts may arise which are semantically unnecessary. Unfortunately, false-sharing is a time dependent phenomenon and it may turn into true-sharing in some later time interval. Our object relocator is able to resolve such situations by relocating involved objects to other free memory pages, see Fig. 6.

Of course relocating objects tends to store only one single object per page destroying locality benefits of the page granularity. Currently we are developing policies to cluster scattered objects using the relocation facility to improve locality by true-sharing.

5 Evaluation and Conclusion

The table on the next page shows preliminary statistics of the characteristics of the distributed heap (object types and backpacks) and the start-up time of single nodes. The computers are booted in numerical order. System objects must not be invalidated because they are essential for system operation (e.g. the page-fault handler) and thus require special treatment by the memory management. Double numbers *(<normal count>/<system count>)* appearing in Table 1 indicate normal and system objects. Column 4 shows the values for three nodes running the oberon like GUI.

The first section shows the heap content after start-up of the system consisting of ~40.000 lines of OS and driver code. In the second section the backpacks statistics are presented for the respective heap objects. Most of the objects do not require a backpack as they are referenced by less than three instances only. Joining nodes allocate new instances with possible new backpacks but classes and methods are shared through the DSM. The following section shows the usage of backlinks (three available per object). Most of the objects need only a single backlink. The inheritance section lists the inheritance depth of classes. Dynamic methods are replicated in method jump tables of subclasses and the code-segments are shared by subclasses. As a consequence code-segments may be referenced by more than three classes and may then require a backpack. Fortunately, most of the class inheritance hierarchies have less than three levels. The final part shows startup time with backpacks and an OS version compiled without bookkeeping of references. Startup time of Node 1 is slower because it creates the initial DSM heap. Node 2 and 3 only join and fetch desired objects from the already running heap.

These figures support our hypothesis that the backpack scheme can be implemented without excessive penalties in memory usage and execution time. Together with bi-directional runtime structures generated by our Java compiler it is an elegant foundation for incremental garbage collection and object relocation. The latter is essential for heap compaction and resolution of false-sharing situations.

Table 1. Measurements for system startup

	Node 1 (P4 2,54 Ghz)	**Node 2** (Athlon XP 2,0)	**Node 3** (Athlon XP 2,2)	**Running GUI**
Objects				
- total	7339	7673	7987	8556
- instances	2032 / 253	2239 / 263	2444 / 273	2727 / 273
- methods	1291 / 582			
- classes	170 / 58			
- backpacks	1009 / 0	1071 / 0	1133 / 0	1270 / 0
- arrays	1710 / 234	1765 / 234	1802 / 234	1951 / 234
Backpacks				
- total	1009	1071	1133	1270
- instances	550 / 253	583 / 263	599 / 273	745 / 273
- methods	136 / 0			
- classes	184 / 0			
- arrays	139	168	197	201
Backlinks				
Only 1	4556 / 198	4636 / 205	4715 / 212	5097 / 212
Two	379 / 0	339 / 0	338 / 0	362 / 0
Three	105 / 0	137 / 0	98 / 0	123 / 0
>three	276 / 1	316 / 1	387 / 1	398 / 1
Inheritance				
Only 1	4580 / 535	4865 / 537	5166 / 541	5655 / 541
Two	1330 / 588	1357 / 591	1384 / 594	1392 / 594
Three	19 / 6	20 / 9	21 / 12	57 / 12
>three	283 / 0	284 / 0	285 / 0	287 / 0
Startup time				
Backpacks	803 ms	350 ms	235 ms	-
no backp.	794 ms	350 ms	235 ms	-

References

[1] Plurix Homepage: `http://www.plurix.de`

[2] N. Wirth and J. Gutknecht, *Project Oberon - The Design of an Operating System and Compiler*, Addison-Wesley, 1992.

[3] J.L. Keedy and D.A. Abramson, "Implementing a large virtual memory in a Distributed Computing", *Proceedings of the Hawaii International Conference on System*, Hawaii, USA, 1985.

[4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", *Proceedings of the International Conference on Parallel Processing*, 1988.

[5] M. Pizka and C. Rehn, “Heaps and Stacks in Distributed Shared Memory”, *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, Ft. Lauderdale, Florida, USA, 2002.

[6] M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 1999.

[7] A.C. Myers, “Bidirectional Object Layout for Separate Compilation”, *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Texas, USA, 1995.

[8] M. Schoettner, *Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher*, Dissertation, Universität Ulm, 2002.

[9] F. Le Fessant, “Detecting distributed cycles of garbage in large-scale systems”, *Principles of Distributed Computing*, Rhodes Island, August 2001.

[10] J.L. Peterson and T.A. Norman, “Buddy systems”, *Communications of the ACM,* 20(6), 421–431, 1977.

[11] D. Mosberger, “Memory Consistency Models”, *ACM Operating Systems Review*, 27(1), 18–26, January 1993.

[12] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, „Optimistic Synchronization and Transactional Consistency”, *Proceedings of the Workshop on Distributed Shared Memory on Clusters*, IEEE International Symposium on Cluster Computing and the Grid, Berlin, Germany, 2002.

[13] R. Goeckelmann, M. Schoettner, M. Wende, T. Bindhammer, and P. Schulthess, „Bootstrapping and Startup of an object-oriented Operating System”, *European Conference on Object-Oriented Programming – Workshop on Object-Orientation and Operating Systems*, Malaga, Spain, 2002.

[14] W.J. Bolosky and M.L. Scott, “False sharing and its effect on shared memory performance”, *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, San Diego, USA, 1993.

Caste: A Step beyond Object Orientation

Hong Zhu and David Lightfoot

Department of Computing, Oxford Brookes University
Oxford, OX33 1HX, United Kingdom
{Hzhu,dlightfoot}@brookes.ac.uk

Abstract. The concepts of object and class have limitations in many situations, such as where inheritance does not offer natural solutions and design patterns have to be applied at a cost to polymorphism, extendibility and software reusability. Proposals have been advanced to generalize the concepts of object and class, such as in Active Oberon. We adapt the concept of agents to software engineering, propose a new concept called caste – a sort of 'dynamic class' of agents, and examine how it helps solve common problems in object orientation.

1 Introduction

Agent technology has mainly been seen as an aspect of artificial intelligence, though it is increasingly seen as a viable approach to large-scale industrial and commercial applications [1]. In our work we disregard the more evidently anthropomorphic aspects of agent (intention, desire, belief, etc.) and concentrate on those aspects that pertain to software engineering.

The concept of an autonomous agent with its own encapsulated data and procedures can be seen as an extension of the object of OO programming. One of the main features of agents lacking from object-orientation is the encapsulation of process with the state and operations, and the explicit description of environment. The language Active Oberon (Zonnon) [2, 3, 4] includes a means of encapsulating a process as an object body. Another weakness of object orientation is the static nature of class structure. In this paper, we investigate how the concept of class can be naturally generalized by a new concept called *caste*, which is roughly a 'dynamic class' of agents.

2 Limitations of Class Structure

The concepts of class and inheritance are central to object orientation. The inheritance hierarchy of classes enables polymorphism and extendibility in object-oriented programs, and hence helps improve software reusability. However, single inheritance imposes a tree-like hierarchy on classes that may not match reality well. Multiple inheritance, or the generalized definitions of Active Oberon, reduce this restriction. However, a further restriction is that an object must at all times belong to one and the same class. This can pose difficulties if the relationship between an object and its class needs to be more dynamic. For example, a personnel information management system of a university may preclude a student from being enrolled both as an undergraduate and as a postgraduate, but it is quite common at our university for a staff member to be simultaneously a student. If the lifetime of the system is long enough then it will need to model the change of status, for example, when an undergraduate successfully graduates and wishes to continue as a postgraduate. This cannot be naturally modeled by inheritance relations between classes.

The use of an appropriate 'design pattern' [5, 6] helps overcome these difficulties, but at a cost to polymorphism and extendibility. Such difficult design problems inspire us to search for programming-language facilities that can lead to natural solutions while retaining the advantages of object orientation.

3 Agent and Caste

We define agents as active and persistent computational entities that encapsulate data, operations and a behavior protocol and are situated in their designated environments [7]. Here, data represents an agent's state. Operations are the actions that an agent can take. Behavior protocols are rules that determine how the agent changes its states and performs actions in the context of its environment. Each agent has its designated environment, which explicitly specifies a subset of the other agents in the system whose behaviors and states will affect the agent's behavior. By encapsulation, we mean that an agent's state can only be changed by the agent itself, and an agent decides its state changes and actions according to its own behavior protocol. As argued in [7, 8], objects are a degenerate form of agents. The structure of an agent is as follows.

- *Agent name* is the identity of the agent, which can be created as a member of several castes.
- *Environment description* indicates a set of agents whose visible actions and states are visible to the agent. As in the formal specification language SLABS [7], an environment description can be in one of the following forms. (1) *Agent-name*: which means that the agent of the name is in its environment; (2) *All*: *Caste-name,* which means that all the agents in the caste are in its environment; (3) *Variable: Caste,* which means that a specific agent in the caste is in its environment, but the agent may change from time to time. Notice that, environment description differs from the import/export facility in that it describes what kind of agents in the system it will interact with at run-time, which cannot be determined at compilation time. It is one of the most important features of agents that distinguish them from objects, including active objects. The environment of an agent changes when other agents join or quit a caste, if the caste is specified as a part of the agent's environment. An agent also changes its environment by joining a caste or quitting from a caste, or changing the values of its environment variables.
- *Variable declarations* define the state space of the agent. It can be divided into two parts. The *visible part* consists of a set of public variables whose values are visible, but cannot be changed by other computational entities in the environment. The *internal state* consists of a set of private variables and defines the structure of the internal state of the agent, which is not visible by other entities in the environment.
- *Action declarations* are in the form of a set of procedure declarations, which defines a set of operations on the internal state and forms the atomic actions that the

agent can take. Each action has a name and may have parameters. An action can be one of two types. When executed, a *visible action* generates an event that is visible by other agents. *Internal actions* can only change the internal state, but generate no externally visible event when executed.

- *Body* is an executable code that forms the agent's behavior protocol. As mentioned above, agents are active computational entities. Their dynamic behaviors can be described by the pseudo-code in Fig. 1.

Usually, the body code of an agent perceives the visible actions and states of the agents in its environment, and decides on what action to take according to the situation in the environment and its internal state. An agent can (1) take a visible or internal action; (2) change its visible or internal state; (3) *join* into a caste or *quit* from a caste. An agent's action is not driven by 'method calls' from the outside. This distinguishes agents from active objects.

```
BEGIN
    Initialization;
    Loop Body-code Endloop;
END
```

Fig. 1. Agent's behavior

Caste is a new concept and a language facility first introduced in SLABS [7]. It is a natural evolution of the concepts of class in OO and data type in procedural languages. Just as a class can be seen as a set of objects with the same pattern of data and methods (procedures), a caste is a set of agents with the same pattern of data, methods, behaviors and environments. The concept of a caste contrasts with the class, however, in that an agent may *join* a caste or *quit* from it at runtime whereas an object is all time an instance of one class. This more general view overcomes the above problem in object orientation. The structure of a caste declaration is given in Fig. 2.

```
CASTE name OF caste-names;
    ENV environment-descriptions;
    VAR variable-declarations;
    ACTION action-declarations;
    INITIAL (parameters): Statement;
BEGIN
    Statement (* body code *)
END name.
```

Fig. 2. Structure of castes

A formal definition of the semantics of castes can be found in [9]. Fig. 3 shows an example of caste declaration, *Persons*. In Fig 4, castes *Undergraduates*, *Postgraduates*, *PhD_Students*, and *Faculties* are declared as sub-castes of *Persons*; some details are omitted for the sake of space. An agent of caste *Persons* can join the caste *Undergraduates*. By doing so, the agent obtains some additional state and environment variables defined in the caste Undergraduates, i.e. the *Personal_ Tutor* and *Student_ID*. The agent can then quit from the *Undergraduates* and join the *Postgraduates*. Consequently, the agent loses state variable *Student_ID* and environment variable *Personal_ Tutor*, and obtains additional state and environment variables *MSc_Student_ID* and *Supervisor*. Subsequently, it can quit from the caste and join *PhD_ Students*. However, it can join Faculties without quit from *PhD_Students*.

```
CASTE Persons;
    ENVIRONMENT All: Persons;
    VAR  PUBLIC   Surname, Name: STRING;
         PRIVATE  Birthday:DATE;
    ACTION PUBLIC Speak(Sentence: STRING);
    INITIAL (SN, N: String, BD:DATE):
        BEGIN
            Surname := SN; Name := N;
            Birthday := BD;
            Speak("Hello, World");
        END;
BEGIN
    … JOIN(Undergraduates); …
END Persons.
```

Fig. 3. An example of caste

4 Conclusion

In this paper, we examined the concept of agent and caste as a language facility to extend object orientation. We are aware of a variety of approaches to the problem we addressed in this paper. We believe that the approach proposed here looks promising to overcome the weakness of static object-class binding in object orientation in a nice and natural way.

We are working towards developing a programming language using agents and castes as a core language facility. The intention is to retain the advantages of object orientation and build on its success while offering more powerful and natural means of expression for solutions to commonly occurring problems in software design. We are investigating the implementation of such a language facility, for example, through active objects.

```
CASTE Undergraduates of Persons;
    ENVIRONMENT Personal_Tutor: Faculty;
    VAR   PUBLIC   Student_ID: Integer;
    INITIAL (PT: Faculty):
        BEGIN  Personal_Tutor := PT; END;
BEGIN
    … QUIT(Undergraduates);
        JOIN(Postgraduates); …
END Undergraduates;
CASTE Postgraduates of Persons;
    ENVIRONMENT Supervisor: Faculties;
    VAR PUBLIC MSc_Student_ID: Integer;
    INITIAL …;
BEGIN
    … QUIT(Postgraduates);
        JOIN(PhD_Students); …
END Postgraduates;
CASTE PhD_Students of Persons;
    …
BEGIN     … JOIN(Faculty); ….
END PhD_Students;
CASTE Faculties of Persons;
    …
END Faculties;
```

Fig. 4. Examples of subcastes

References

[1] Wooldridge, M., Weiss, G., and Ciancarini, P., eds. Agent-Oriented Software Engineering II. LNCS, Vol. 2222, 2002: Springer.

[2] Gutknecht, J., Active Oberon for .net, White Paper, June 5, 2001. Available at URL: http://www.bluebottle.ethz.ch/oberon.net/ActiveOberonNetWhitePaper.pdf

[3] Reali, P., Active Oberon Language Report, March 14, 2002. Available at URL: http://bluebottle.ethz.ch/languagereport/ActiveReport.pdf

[4] Gutknecht, J., Zueff, E., Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET, Available at URL: http://www.bluebottle.ethz.ch/Zonnon/papers/OOPSLA_Extended_Abstract.pdf

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns,: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[6] Grand, M., Patterns in Java, Vol. 1: A Catalog of Reusable Design Patterns Illustrated with UML, Wiley, 1998.

[7] Zhu, H., SLABS: A Formal Specification Language for Agent-Based Systems. International Journal of Software Engineering and Knowledge Engineering, 2001. 11(5): pp. 529–558.

[8] Zhu, H. The role of caste in formal specification of MAS. in Proc. of PRIMA'2001, Taipei, Taiwan, 2001, Springer, LNCS 2132, pp.1–15.

[9] Zhu, H., Representation of Role Models in Castes, Technical Report DoC-TR-03-02, Dept of Computing, Oxford Brookes Univ., UK, 2003. (*Submitted to MATES'03*)

Distributed XML Objects

Paul Roe

Queensland University of Technology, Australia
p.roe@qut.edu.au
http://www.fit.qut.edu.au/~roe

Abstract. XML and web services are set to revolutionise the Internet. However there is currently an impedance mismatch between XML and traditional programming languages such as Java and C#. This can lead to difficulties implementing web services using these languages. In this paper we discuss this issue and investigate a particular solution, using the XML query language, XQuery, to implement web services. This leads to a system comprising distributed XML objects.

1 Introduction

Web services are touted as an enabling technology for the next generation Internet. Despite the massive amounts hype and opposing scepticism, web services are starting to be used in real applications. Along with this developers are realising that there is more to authoring web services than writing a class and automatically generating stubs and web service meta-data. Unlike Java RMI, DCOM and CORBA, web services are not a technique for remote object access, web services are best viewed as infrastructure for XML messaging.

Web services such as SOAP support XML based messaging and RPC across protocols such as HTTP. Thus for a traditional language such as Java or C# to use or provide web services, programming language objects must be mapped to XML. Unfortunately programming objects and XML are rather different: programming language objects can contain references, be arrays, or have private fields; XML has not direct counterparts to these. XML contains elements and attributes, default and optional values, and programming languages have no direct counterparts for these. There are a variety of solutions to this problem including:

– Use just the common subset – this is a restriction on both the XML and programming language types that are used. However, there is no specification of the subset or a way to enforce its use, and the more languages which interoperate the smaller the subset becomes.
– Implement a mapping between XML and programming language types. This is difficult to do automatically, and can give rise to unnatural XML or programs. Since web services should be designed first, for reasons of efficiency, interoperability and evolution, it is the program that is likely to suffer. Often some hand crafted mapping code will be required.

– Use DOM, a faithful model of XML, in programs. This is rather indirect and clunky, but nevertheless a solution often adopted.
– Extend programming languages to support XML. This is a promising approach; two rather different examples of this are: the Xtatic project which is investigating incorporating statically typed XML into the C# programming language [3,2] (and also [16]), and native XML scripting for ECMAScript (dynamically typed) proposed by BEA [11] and others.
– Use a native XML programming language such as XSLT or XQuery.

The latter is the subject of this paper; a system is proposed which enables objects, represented as XML documents, to be exposed and acted upon by web services implemented using XQuery. The main contribution of this paper is to propose an XML object system for performing distributed computing using purely XML.

2 XML Objects

An XML object is an XML document, represented by a DOM; this may be persisted as a file or as part of a database. URIs are used to reference objects; thus URIs are used, as intended, for identifying resources. Object methods are simply web services. XML objects may be dynamically created, thus unlike traditional web services, web services are dynamically provided. This also leads to a REST like [1] style of web architecture where URIs denote individual resources (objects) to be communicated with rather than communications endpoints.

We want to implement web services using XQuery because XQuery is designed for manipulating XML, is strongly typed, being standardised by W3C, and is easier to use than XSLT. To do this web services must be mapped to XQuery operations. This is achieved through an implementation mapping document that specifies the interfaces (WSDL port types) that an object implements and which binds interface operations to XQueries. The mapping document is used for dispatching web services to XQueries and for generating WSDL descriptions of the services an object supports, see Fig. 1.

In addition to SOAP web services, some specialised interfaces are also supported, for example to generate XHTML for user interfaces.

The combination of XML objects and web services yields a COM like object model, where objects implement named interfaces (WSDL port types). There is no other form of polymorphism.

3 Implementing Web Services Using XQuery

XQuery is purely functional which means that whilst it can query data it cannot update data (yet) or perform any imperative actions. To solve this problem we use an idea from Haskell, each XQuery results in an XML action document which describes an imperative action to perform e.g. an object update. A list of actions is shown below:

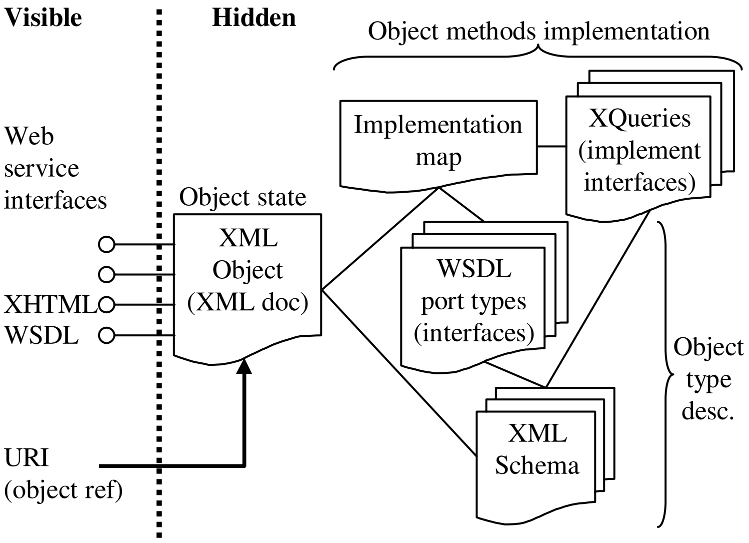


Fig. 1. XML object implementation

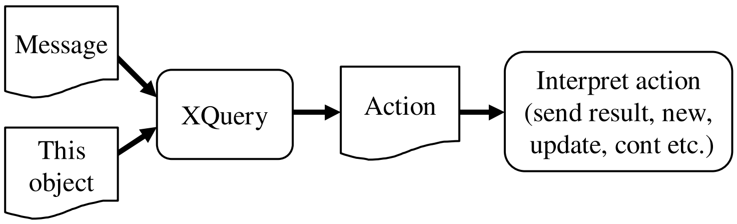


Fig. 2. Send message processing

Result: this indicates the result to be returned from a web service invocation.

Update: this specifies a new document to replace the old one; in principle deltas could also be specified.

New: this creates a new object at a new URL on the same host. The initial object state and implementation map are specified.

Copy: this copies the current object to a different URL on the same host.

Send: this sends an XML message; the received result is supplied to the (optional) specified continuation.

Continuation: this specifies an explicit continuation to run after the action, e.g. update, has completed.

The processing of a web service message is shown in Fig. 2. Actions generated by an XQuery are interpreted by a small program. All messages are automatically wrapped and unwrapped with SOAP envelopes.

Whilst an XQuery is being processed the associated object is locked so any update action is guaranteed to be atomic. Since sends typically correspond to

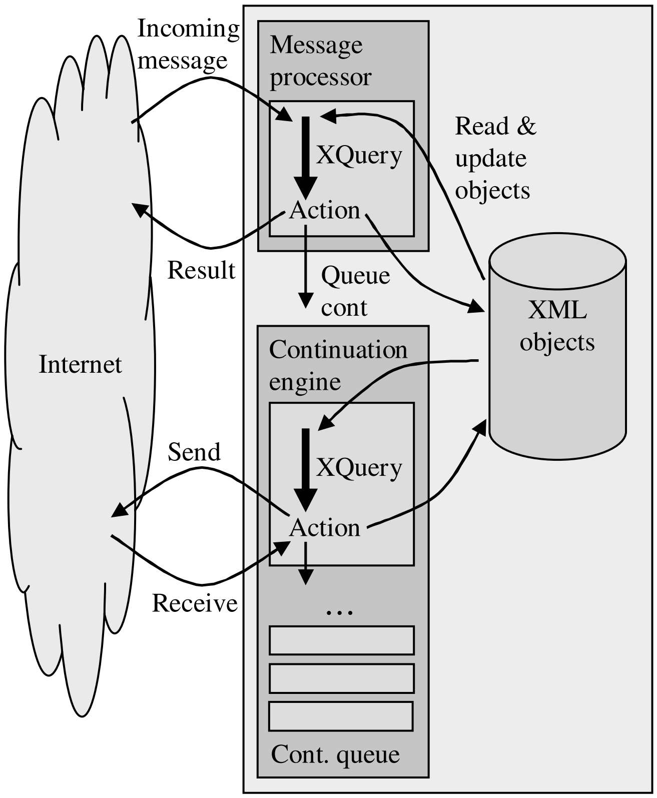


Fig. 3. XML object processing

an HTTP request, they must be short, and non-blocking. This is achieved by delaying sends to after message processing and this has the added benefit of preventing certain forms of deadlock due to object re-entrance.

Send actions are delayed by treating them as continuations, which are queued and processed by a separate thread from the message processing thread. Continuations are used to implement complex behaviours, including notifications. Thus the XML object engine comprises a message processor for incoming messages and a separate thread and associated queue for sends and continuations, see Fig. 3.

4 Discussion

4.1 Implementation

A prototype implementation has been constructed using Java, the Fraunhofer IPSI-XQ XQuery implementation [13], and the Tomcat web server. All XML processing was performed either by the Java XQuery engine or by DOM manipulation; persistence of web objects was not implemented, nor was any type (schema) checking outside of the XQuery system. Several toy examples, including part of a paper review system, were constructed.

4.2 The Model

The model is simple but also limited. The existence of XML publishing frameworks such as Cocoon [6] and the new Microsoft Office program InfoPath [9] gives credence that some categories of applications should benefit from such an XML approach. Clearly more experience with writing applications is required.

Currently object location, discovery, and migration are all open issues. How should objects be arranged and discovered? Can we migrate objects from one machine to another?

There are questions concerning typing. Unfortunately given that there are many outstanding issues regarding typing XQuery, these issues cannot be resolved yet. For example is XQuery's static typing strong enough for effective programming?

Support is needed for exception handling from SOAP faults. Various approaches to this are possible. Of great importance are security and transactions, we wish to leverage the developing WS-Security and WS-Coordination standards. This involves manipulating SOAP extensible header information; however this needs to be done implicitly.

Generating user interfaces with XHTML works but it is not very elegant. XForms looks like a good candidate replacement once it is finalised and implementations become available [7].

4.3 Efficiency

The XQuery language is not particularly efficient; however it is comparable with a scripting language and has the potential for better performance if compiled e.g. the Kava system [12]. Given the intended use of XQuery for querying large amounts of data we expect optimising XQuery implementations to become available.

Some of the systems inefficiency is from the DOM representation of XML; in the future more efficient implementations of XML will become available. An obvious source of inefficiency is the update action; this will change once update becomes part of XQuery.

4.4 Related Work

There are two projects similar to that described here: Active XML [8] and XL [14], both utilise web services and XQuery. Active XML takes a database approach, XML documents are stored in a database and are acted upon by web services. Web services are implemented by XQuery operations embedded within

XML documents. The system is akin to a database with analogies to triggers and stored procedures, rather than a distributed object system.

XL is a high level imperative language for writing web services, which embeds XQuery. It is more like a workflow language rather than a distributed object oriented language. For example it supports operations such as events, logging and retry actions. There are numerous other proposals for XML scripting languages, but they do not support distributed objects, e.g. [15].

Acknowledgments I wish to thank Patrick Lehti for his support with the Fraunhofer IPSI-XQ XQuery implementation. This work was carried out whilst the author was on sabbatical at INRIA Sophia Antipolis.

References

1. R. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD Thesis, University Of California, Irvine, 2000.
2. B.C. Pierce, Xtatic homepage, `http://www.cis.upenn.edu/~bcpierce/xtatic/`
3. H. Hosoya and B.C. Pierce, XDuce: A typed XML processing language. ACM Transactions on Internet Technology, 2002.
4. W3C, XQuery, `http://www.w3.org/TR/xquery/`
5. D. Chamberlin, XQuery: An XML query language, IBM Systems Journal, Vol 41, No 4, 2002
6. Apache Software Foundation, Cocoon, `http://xml.apache.org/cocoon/`
7. W3C, XForms, `http://www.w3.org/MarkUp/Forms/`
8. S. Abiteboul, O. Benjelloun and T. Milo, Web services and data integration, International Conference on Web Information Systems Engineering, Singapore, Dec. 2002
9. Microsoft, InfoPath, `http://www.microsoft.com/office/preview/infopath/`
10. W3C, Web Service Activity, `http://www.w3.org/2002/ws/`
11. BEA, Web Logic, `http://www.bea.com/`, (ECMA Script XML extensions `http://edocs.bea.com/workshop/docs70/help/guide/xmlmap/conHandlingXMLWithECMAScriptExtensions.html`)
12. Per Bothner, Qexo homepage, The GNU Kawa implementation of XQuery, `http://www.gnu.org/software/qexo/`
13. Fraunhofer IPSI, IPSI-XQ - XQuery homepage, `http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html`
14. D. Florescu, A. Grunhagen and D. Kossmann, XL: An XML Programming Language for Web Service Specification and Composition, WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA. (`http://www2002.org/CDROM/refereed/481/index.html`)
15. DecisionSoft Ltd, XMLScript, `http://www.xmlscript.org`
16. E. Meijer and W. Schulte, Unifying Tables, Objects, and Documents, submitted to OOPSLA 2003. (`http://research.microsoft.com/~emeijer/`)

Programming Education: A Russian Perspective

Fyodor V. Tkachov

Institute for Nuclear Research of Russian Academy of Sciences
Moscow 117312, Russia
http://www.inr.ac.ru/~info21/

Abstract. A unique combination of properties makes the Oberon family of languages an ideal platform for algorithm design work in computational sciences as well as for a systematic general programming education. The project Informatika-21 builds on the strong Pascal tradition in Russia and purports to promulgate Oberon in Russian education with an ultimate goal to establish a system under which high school and university students would be universally and systematically exposed to the fundamentals of programming similarly to how they are exposed in Russia to the fundamentals of mathematics.

1. The educational project Informatika-21 [1] has grown out of (i) a discovery that programming languages from the Oberon family[1] hit the nail on the head in regard of the problems that plague programming in modern physics [6], (ii) a realization that the design decisions behind Oberon adequately address universal problems, thus providing an excellent foundation for a general programming education for professionals in the fields beyond CS such as physics, engineering, chemistry, linguistics [17], etc.

Specifically, in physics as practiced at CERN etc., the following trends are observed:

— Even run-of-the-mill projects increasingly require the use of two or more conventional specialized and non-interoperable programming systems. The problem of symbolic/numeric interface is an old example. A modern one is a web service based on libraries of fortran and symbolic manipulation codes, managed with a database and glued together by Java [7].

— Specialized systems naturally evolve towards incorporating core features of a general purpose language. The result is a duplication of this core with sometimes strange variations and gaps[2] across many non-interoperable systems (cf. the above example).

— There is a trend towards large (100-1000 participants), long-lived (on the order of 20 years) yet volatile (students and PhD's joining and leaving), loosely organized international collaborations. One cannot rely on programming gurus only: Involve

[1] For the present discussion the differences between the original Oberon [2], Oberon-2 [3], Component Pascal [4], Active Oberon [5], etc., are mostly inessential.

[2] For instance, absent or rudimentary sets of standard control structures in the symbolic manipulation systems (such as SCHOONSCHIP [16] and its derivatives) that are highly popular among physicists.

ment of rank-and-file physicists in software development is necessary. Enforcement of standards such as language subsets proves to be problematic.

— Even rank-and-file physicists-programmers should be familiar with elements of large scale software design.

— An inflexible division of labor between programmers (even with a physics background) and physicists proper may be detrimental for the project's success: the programmer may agonize over obstacles that could be circumvented by minor changes at the level of formalism.

— Vice versa, a competence in mapping formulae to data structures and algorithms helps to focus the theorist's work in a surprisingly productive fashion (cf. the invention of what is considered to be the most popular family of algorithms for one of the most important classes of calculational problems in theoretical particle physics [8]). The thought patterns of algorithm design must become part of the theorist's problem solving arsenal.

— Systematic programming is becoming physicists' everyday instrument, and we should be able to do it in as casual a manner, without becoming full-time programmers, as we differentiate and integrate without becoming mathematicians. Unfortunately, such a casual use is almost impossible with the most popular SD systems.

The above trends have been manifest in the physics community but one cannot fail to notice similar trends in other scientific fields.

CERN made an attempt to create a common SD platform for its Large Hadron Collider mega-project (2006-2020) by adopting C++ as a standard back in 1994. Needless to say, C++ cannot be a half way adequate solution for the listed problems because of its well-known critical design defects and extravagant complexity. In fact, there is anecdotal evidence of splits in physics collaborations in regard of the use of C++, and a number of physicists are hesitant to switch to C++ and choose to stay with fortran. The failure of C++ is visualized, so to say, by the CERN-developed data analysis and visualization framework ROOT [18], which is notorious for segment violation crashes.[3]

Clearly, a lucid minimalism is absolutely key if the programming language is to synergize in physicists' brain space with all the required physics and mathematics.

2. The present author has been researching the subject of calculational algorithms for theoretical physics of elementary particles since 1978 both at the level of quantum field theory formalism and at the level of implementation of the resulting algorithms. On the programming side, my last significant completed project dealt with experimental data processing [10], another one was of a numerical nature [9], and the bulk

[3] As of March 2003, ROOT running on Linux would cause three segment violation show stoppers in less than five minutes of a theorist's demonstration of his cosmic rays work where all one needed was to visualize positions of various cosmic rays sources on a sky map. Another example: a highly experienced and respected programmer changed her specialization in order not to deal with the ever crashing C++-based software: she stated it crashes even worse than how Soviet-made clones of IBM mainframes would require a reboot every 15-30 minutes back in the 80s.

of effort goes into large-scale theoretical symbolic manipulation ([8] and refs. therein). The large, "industrial" scale of such calculations (CPU time on high end hardware measured in months; for longer calculations it is more reasonable to upgrade hardware) makes run-time efficiency a central consideration. A possibility to design data structures fine-tuned for a specific class of problems and to compile significant portions of the code can boost efficiency by many orders of magnitude compared even with narrowly specialized conventional computer algebra systems (an example in [11] shows an efficiency increase measured by a factor of fifty million). Software evolvability is of paramount importance here due to all the experimentation with data structures and algorithms needed to arrive at satisfactory (not to say efficient) programs. This concerns both the level of code (static safety etc.) and a proper design (a careful stratification, encapsulation, etc.).

These considerations have eventually led me to the adoption since 1995 of Oberon (specifically, the Component Pascal variation of Oberon-2 as implemented in the BlackBox Component Builder from Oberon microsystems, Inc. [4]) for all my algorithm design work. Oberon/Component Pascal has proved to be a superb choice for the entire range of problems I deal with — from various flavors of symbolic manipulation (large-scale algebra, graphs, combinatorics) to purely numerical applications (e.g. multidimensional optimization and adaptive Monte Carlo integration). The type safety, modularity, automatic memory management, and a fast compiler in an unencumbered development environment capable of running on any old PC or notebook, almost anywhere, result in a surprisingly productive development and experimentation cycle even for purely numerical projects to be eventually ported to fortran.[4]

In fact, for creative algorithm design work, especially where run-time efficiency is a top concern, Oberon transcends the role of a mere "tool" and becomes a true brain extender.

3. A few years of failed attempts to find collaborators for projects based on Oberon (despite sincere expressions of interest: my fellow theorists are too busy debugging their fortran, C++, etc. codes; and this is less of a joke than one might think), convinced me to start an experimental course of modern programming techniques at the Physics Department of the Moscow State University in the Spring semester of 2001. The experience quickly revealed that:

— Various courses of the regular programming curriculum (which spans the first two years and includes the usual introductory courses, a practicum, and a physics modeling mini-project) use various programming platforms (from MathLab to C++) according to a particular professor's preferences.

— Students waste mental capacity on learning syntactic peculiarities of archaic systems rather than mastering the fundamentals of systematic program construction (step-wise refinement, loop invariants, interfaces as contracts, etc.).

[4] It is somewhat hard to quantify the productivity increase resulting from the simplicity and safety of Oberon. As an example: it took less time to perform all the algorithmic experimentation in Component Pascal to develop an efficient algorithm for an optimization problem in O(2000) dimensions reported in [10] (about four weeks), than to subsequently port it to fortran. Moreover, a rounding-errors-related problem was only resolved by switching back to CP after a failure to resolve it directly in fortran.

— Completely missing from the courses are the key notions of programming in the large (software evolvability, interfaces, patterns, …).

In short, the complexity and disparity of various programming platforms used throughout the programming curriculum have a hugely adverse effect on what students' actually learn about programming.

On the other hand, the lucid minimalism of Oberon/Component Pascal made it possible to pack into a semester-long lecture course (formally similar to a standard introductory course) a range of modern programming techniques including elements of Dijkstra's discipline of programming, the basic dynamical data structures, essential OOP, COP and basic patterns (factory, carrier/rider, separation of interface from implementation), basics of interactive graphics (MVC). The course content could be further enriched as experience and the library of program examples is accumulated. The duration of only one semester (16 lectures) is dictated by extraneous circumstances and is, of course, somewhat too restrictive: it would be better to devote a full semester to programming in the small and another one, to programming in the large. Yet even a one-semester course is possible (it acquired a pretty stable form towards the third edition) thanks to the use of Oberon/Component Pascal as the foundation language.

The lectures are based on program examples of increasing complexity; their subjects are correlated with the main curriculum (math courses, physics lab practicum). The course efficiency is greatly increased by distributing printed notes (with complete program sources along with screenshots, excerpts from documentation and various comments, slogans and admonitions) to students before each class. A textbook — although a useful supplement and necessary for a wider distribution of the course material — would be more limiting in regard of the material presented, and a computer presentation — although sometimes irreplaceable — would limit students' activity in adding their own notes.

The course attracted both students dissatisfied by the standard programming courses for various reasons as well as students with a practical experience in programming but seeking a systematic view of the subject, and also the faculty looking to enhance personal productivity.

4. A correct positioning of the course was seen to be crucial in view of the religious nature of debates around programming languages, and is currently as follows:

— Oberon/Component Pascal provides the least painful introduction into modern programming. After learning the basics of program design in Oberon, switching to other popular languages reduces to learning inessential (syntactic etc.) details in the existing special courses traditionally devoted almost exclusively to such details (fortran, C, C++).

— Oberons are powerful productivity tools to take the sting out of students' chores such as data processing in physics labs, the practicum in numerical methods, as well as for research assignments and subsequent individual research (we are talking here about research in physics etc., not computer science). The latter assertion is supported by the fact of the course attendance by the faculty.

— Even when forced to program in other languages, following a systematic programming style learned with Oberon mitigates their deficiencies. Students indeed reported productivity gains and a reduced debugging agony, e.g. in a practicum on numerical methods, with first developing their assignments in Oberon and finally porting them to C as required.

— Critically important in the Russian context is that with Oberons, students are not limited by the hardware at their disposal: some students only have unrestricted access to an 486. But this proves sufficient to fully experience modern programming with Oberons.

— Last but not least: Sun's Java and Microsoft's C# — the two languages vying for dominance in the business software industry — have more in common with Oberon than with their syntactic predecessors C and C++. This can be interpreted as signifying a consensus of the software market leaders in regard of the core set of essential features that a modern general purpose programming language must possess — counting a (relative) minimalism as a feature. Of the three languages, Oberon remains best suited for numerical applications (an important argument e.g. in physics), and — owing to its superbly clean design — the easiest introduction into what can be called *the standard paradigm of modern programming*. This point was nicely illustrated by the fact of students obtaining well-paid part-time jobs involving programming in C# after attending the first edition of the course.

5. A serious problem has also become evident. Namely, a number of students enter the university with already a non-negligible experience in programming (often from high schools with advanced curriculum in physics and math). Unfortunately, it is either the old Pascal, Borland's Pascal or C/C++ — taught by rarely competent school teachers who mistake, as usual, complexity for power. Such students are reluctant to attend what appears to them an introductory course, opting to invest their spare time into commercial programming. And bad habits in programming formed by inadequate training are hard to eradicate. (Note that analogy with sports where proper techniques is essential for top achievements sometimes works well to motivate such students to undertake the effort. The analogy is less superficial than it may seem at first glance.)

As soon as the problem of programming education at the high school level emerged, an experimental high school course was initiated in the Fall of 2001 — motivated partly by a necessity to ensure a flow of well-trained students, partly by an idealistic view of the general importance of the problem.

6. A few remarks are in order on the subject of education system in Russia. The disruptive economic reforms cause discussions of how to capitalize on Russia's strengths in basic research and education in a post-industrial transformation. Since the educational reform of 1871 following the 1866 attempt on the life of Czar Alexander II, a strong emphasis was placed on mathematics in the curriculum (curiously, along with classical languages) as an antidote to propagation of freethinking [12]. After 1917, the education effort encompassed the entire population with classical languages replaced by natural sciences. The education system with a solid and universal mathematics curriculum proved to be an excellent foundation for the development of the aerospace industry and the schools of mathematics and theoretical physics.

A natural idea then is to supplement the strong mathematical tradition with a systematic education in programming. Indeed, IT analysts note that in the international off-shore programming market Russia already tends to specialize in mathematics-intensive software (cf. the projects of the Intel Software Development Center in Nizhny Novgorod half-way between Moscow and the Urals [19]). Although the revenue volumes are small compared e.g. with India's software industry, such software is less amenable to automation and its role and value is growing.

7. Implementation of this vision must take into account a number of circumstances specific to post-USSR space:

— Obsolete hardware in most schools and homes will be precluding the use of heavy-weight programming systems for quite a while.

— A poor knowledge of foreign languages in Russia and most other post-USSR republics creates informational difficulties for teachers and students alike; on the other hand, it creates a domain relatively less sensitive to foreign marketing influences.

— A strong Pascal tradition: Pascal (various versions) remains by far the most popular platform to teach programming in Russia. This in part is explained by the popularity of Algol-60 in the USSR university education in the 70s (the first systematic course of programming I took in 1976 was based on Algol-60). That in turn was due to the influence of mathematicians who from the very beginning advocated a systematic approach to programming and realized the importance of the programming profession for the future (the most influential person in this respect was A.P.Ershov [20]; his heritage is very much alive, especially in Siberia where, incidentally, most of the Russian aerospace industry is located).

— An army of relatively well-trained mathematics teachers from whose ranks programming teachers are often recruited, and for whom Dijkstra's discipline of programming and Oberon's regularity would have an appeal. Professional mathematicians are often involved in teaching at high schools with extended curricula in physics and mathematics (a system of such schools is also a heritage of the Soviet era); cf. a textbook with many examples of systematic algorithm construction in Pascal written by a mathematician [13] (incidentally, Oberon is mentioned in the introduction as a potentially more elegant choice).

— A lack of guiding lines and reference points, which informatics teachers keep complaining about. For those originally trained as mathematicians, the Euclidean geometry would be an obvious model, and they would greet a similarly systematic exposition of foundations of programming.

— The Pascal tradition in education correlates with the industrial use of Wirthean languages. One example is a sizeable and active Russian community of Delphi programmers [21]. Another is the fact that all the embedded software in Russian communication satellites is being written in Modula-2 [22].

In view of the above, it is inevitable that Oberons should gain a popularity in the post-USSR space. The adoption of Oberon is indeed happening in more than one way. The research and education efforts at the Physics Department of MSU have already been mentioned. In the aerospace industry, there is an interest in Oberon as a natural successor to Modula-2. Since 1998 there exists a web-site [14] authored by

Prof. S.Sverdlov (Univ. of Vologda, some 450 km North of Moscow) to promote Oberon for IT majors at universities;[5] it offers a rich selection of Oberon-related publications in translations into Russian (notice the attention Oberon has been receiving in Russian-language versions of IT publications such as ComputerWeekly and the PC World). Since 1999, the BlackBox Oberon is being promulgated for programming education at the high school level by the Lithuanian Institute of Mathematics and Informatics [24]. Starting the Fall, 2003, the University of Osh (Kirgizia) is implementing an experimental professional programming education program based on Oberons [23].

8. These grassroots tendencies are being shaped via the project Informatika-21 [1] which started as a web site to serve the needs of the MSU course discussed above and then to help spread the information about Oberon together with the high school course material in Russian (most notably, a russification package for the BlackBox version of Oberon including Russian-language versions of compiler messages, instructions for the basic use, simple examples of program development, etc.). The high school programming education space has been chosen as a primary focus because this is where hand-holding is needed most whereas the effect of switching to Oberon should be greatest, long-term.

There has formed a board of advisors representing the aerospace industry, the Russian Academy of Sciences, and the Moscow State University. The project is being helped by volunteer coordinators in Siberia (Dr. A.I.Popkov) where a considerable pool of engineering talent is located, and in Central Asia (Prof. Kubanychbek Tajmamat oolu) where the education problems are similar to Russia's if somewhat exacerbated by demographics but there is more freedom for experimentation due to less bureaucracy. Several dozen copies of the russification package designed specifically for schools have been downloaded from the Informatika-21 site (taking into account that a similar package for professional developers is also available, this gives an estimate for the number of teachers familiarizing themselves with the BlackBox Oberon). Starting the Fall of 2003, new programming courses based on Oberon are due to begin, and another web site is to be put up (the "Leader" Sunday school of programming in Nizhnevartovsk; note how Sunday schools in Russia are about programming rather than religion). Examples of other organizations where Oberon is being evaluated for teaching purposes include the Urals State Politechnical University (Ekaterinburg) and the Siberian Aerospace Academy (Krasnoyarsk). Publication support has been pledged by the MSU Center of Pre-university Education.

The secondary and high school programming curriculum ought to be a simplified, follow-me version of the systematic approach to program construction to be taught at the university level. The project's main goal here is to accumulate a commented collection of exemplary programs correlated with the course material of other subjects. Also, the implications of the powerful automatic memory management feature of Oberon are yet to be explored in the high school teaching of programming. It may be possible, advantageous and fun to introduce dynamical data structures much earlier

[5] Prof. Sverdlov considers it a direct result of using Oberon in education that his students' team made its way into the 27th ACM Int. Collegiate Programming Contest World Finals at Beverly Hills, March 25, 2003.

than is usually done. At the age of, say, 12 kids' very limited knowledge of math is an obstacle for using program examples of a standard kind, whereas learning the basics of list manipulation is fun with standard pictorial representations on the blackboard (e.g. kids' interest is aroused by programs manipulating information on themselves such as a lists-based database of the class with grades, attendance information, etc.). Clearly, Oberon opens non-trivial new possibilities for a more effective (in several respects) course organization.

The described efforts would be much facilitated by an exemplary programming textbook as envisaged by Niklaus Wirth [15]. Fortunately, the Oberon itself is an excellent summary of the basic notions of computer programming and a solid technological foundation on which to build a modern programming education system.

Acknowledgments. I thank Niklaus Wirth for many comments on programming education and Jürg Gutknecht for a constructive advice.

References

1. Project Informatika-21, http://www.inr.ac.ru/~info21/
2. Wirth, N.: The Programming Language Oberon, Software — Practice and Experience, 18 (1988) 671–690;
   Wirth, N., Gutknecht, J.: Project Oberon: the Design of an Operating System and Compiler, ACM Press (1992)
3. Wirth, N., Mossenbock, H.: Oberon-2 Language Report (1992)
4. Oberon microsystems, Inc.: Component Pascal Language Report (2001); http://www.oberon.ch
5. Gutknecht, J.: in Proc. of JMLC'1997. Lecture Notes in Computer Sciences 1024, Springer Verlag;
   Reali, P.: Active Oberon Language Report, http://bluebottle.ethz.ch/languagereport/
6. See e.g. F.V.Tkachov: From Novel Mathematics To Efficient Algorithms, http://arXiv.org/abs/hep-ph/0202033
7. Bardin, D., et al.: Project SANC (2003); e.g. http://arxiv.org/abs/hep-ph/0212209
8. Tkachov, F.V.: Algebraic Algorithms for Loop Calculations, http://arXiv.org/abs/hep-ph/9609429
9. Tkachov, F.V., Manakova, G.I., Tatarchenko, A.F: How Much Better Than Vegas Can We Integrate in Many Dimensions? FERMILAB-CONF-95-213-T (1995)
10. Tkachov, F.V.: Verification of the Optimal Jet Finder, http://arXiv.org/abs/hep-ph/0111035
11. http://www.inr.ac.ru/~ftkachov/projects/bear/dirac.htm
12. Kornilov, A.A.: A Course of Russian History of XIX Century, Vysshaya Shkola, Moscow (1993)
13. Shen, A.: Programming: Theorems and Problems, Moscow (1995); Birkhauser (2000)
14. http://www.uni-vologda.ac.ru/oberon/
15. Wirth, N.: Computing Science Education: The Road not Taken, Opening Address at the ITiCSE Conference, Aarhus (2002)
16. Veltman, M. and Williams, D.N.: Schoonschip '91, http://arxiv.org/abs/hep-ph/9306228

17. Cf. the software package for linguistic studies CHRISTINE designed by the linguist G.Sampson: http://www.grsampson.net/RChristine.html
18. http://root.cern.ch/
19. http://www.intel.com/jobs/russia/sites/nizhny.htm
20. A.Ershov's archive: http://www.iis.nsk.su:81/ershov/english/
21. http://www.delphikingdom.ru/
22. Koltashev, A.A.: presentation at this conference.
23. http://www.kubanych.ktnet.kg/
24. http://www.mii.lt/

Towards an Adaptive Distributed Multimedia Streaming Server Architecture Based on Service-Oriented Components

Roland Tusch

Institute of Information Technology, Klagenfurt University, Austria
roland@itec.uni-klu.ac.at

Abstract. This paper presents an adaptive distributed multimedia streaming server architecture (ADMS) which explicitly controls the server-layout. It consists of four types of streaming server components, which all provide dedicated services in an arbitrary number of instances on an arbitrary number of server hosts. Vagabond2 is used as the underlying middleware for component adaptation. It is shown, how the CORBA-based components have to be declared in order to run on top of Vagabond2. Finally, inter-component dependencies are pointed out, which have to be taken into account during component adaptations.

1 Introduction

Current distributed multimedia servers are monolithic and performance-optimized in order to cope with thousands of simultaneous client requests. However, in case of increasing numbers of clients in heterogenous environments, it is usually not the server, but the network that becomes a bottleneck. Since server implementations are not component-based, they have no chance to cope with this problem and might reject client requests due to network resource shortages. From the client's point of view, one solution to this problem is the usage of a proxy-server. However, this approach has only limited power, as the server has no explicit control over client-side proxies.

Component-oriented programming has shown to be a major improvement in implementing complex distributed systems, allowing to build independent pieces of software that can be reused and combined in different ways. A number of software components for building distributed server applications like distributed web services based on Enterprise JavaBeans, DCOM, or CORBA, exists. Much less effort has been done in building components for distributed multimedia streaming services. There are two key differences between multimedia and non-multimedia services. First, a multimedia streaming service imposes real-time constraints on delivering media data to the clients. Second, the amount of generated network traffic and the time periods for serving clients are usually orders of magnitude bigger than in the case of non-multimedia services. This is why existing components originally built for distributed web services usually cannot be reused for building distributed multimedia services.

Components for real-time server applications also require a middleware that enables for guaranteed or predicted component adaptation times, i.e. times for migrating or replicating a server component from one server host to another. Currently, there is no such middleware system available for operation in internet settings. The first one trying to cope with these issues is a successor of Vagabond2 [1].

This paper explores the building blocks of an adaptive distributed multimedia streaming server architecture, called ADMS [2]. It presents the minimum decomposition of a distributed streaming server architecture, which yet allows to construct flexible demand-oriented streaming services in internet settings. Each of the four distinguished CORBA-based components provides clearly defined interfaces and can be combined in an arbitrary number, allocated on a dynamic number of server nodes. This is achieved by implementing a server component as a so-called *adaptive application* on top of the Vagabond2 middleware. The number of dependent component instances defines a virtual multimedia streaming server cluster. Its size may grow and shrink dynamically over time, depending on QoS requirements derived from client requests, as well as host and network resource usages.

The remainder of this paper is organized as follows. Section 2 gives an overview on component-based architectures, as well as service-oriented middlewares, which allow to dynamically change the location of service-oriented components. The components of the adaptive distributed multimedia streaming server (ADMS) are presented in Sect. 3. In Sect. 4 the issue of dependence management between ADMS components is discussed. Section 5 concludes this work including statements on future work regarding this topic.

2 Related Work

2.1 Component-Based Server Applications

Component-oriented distributed systems are composed of independent, reusable, adaptable and combinable software components. There is a number of standards which enable the building of distributed object systems on the server's side, like OMG's Common Object Request Broker Architecture (CORBA) [3] and its Component Model (CCM), Microsoft's Distributed Component Object Model (DCOM) [4] and .NET framework, and Sun's Enterprise JavaBeans (EJB) model [5]. DCOM, .NET, and EJB do not provide any means for designing a distributed server imposing real-time constraints as required in the case of a distributed multimedia server. CORBA overcomes this drawback by providing Real-Time CORBA [6]. TAO, an implementation of the Real-Time CORBA 1.0 standard providing efficient, predictable, and scalable end-to-end quality of service is presented in [7]. TAO also incorporates an implementation of OMG's Audio/Video Stream Specification [8]. It provides a set of principal CORBA components for end-to-end A/V-streaming using pluggable control and data transfer protocols [9].

2.2 Middleware for Component Adaptation

Middleware providing means to dynamically distribute components or services among a set of server nodes exist in a minor number compared to distributed component-based systems. Three major representative middlewares falling into this category are *Jini* [10], *Symphony* [11], and *Vagabond2* [1]. Jini is a distributed system whose network-centric design lets groups of users share resources over the network, provides access to resources from anywhere while allowing the network location of the user to change. A resource can be implemented as either a hardware device, a software service, or a combination of both. Objects can move from place to place as needed. However, Jini does not provide any means for processing QoS-constrained operations, since it relies on the existence of a network of reasonable speed and latency.

Symphony is a management infrastructure for executing virtual servers in internet settings. A virtual server is a server whose identity is not bound to a fixed physical computer. Depending on the server load it may span a dynamically changing number of server hosts. This is achieved by migrating or replicating a server from one host to another, by instantiating a new service on a certain host, or by evacuating a service from a host. Unlike Jini, its design is based on CORBA technology combined with group communication capabilities, in order to achieve better interoperability, extensibility and reliability. However, the service replication process is not always transparent to the programmer, service replicas may become inconsistent, and the overall system performance is not always good. Symphony was not designed to run multimedia applications, hence it has no support for QoS-constrained services.

The adaptive distributed multimedia streaming server architecture proposed in [2], which serves as the fundament for this work, is based on Vagabond2 [1]. Vagabond2 is also CORBA-based and enables for a dynamic instantiation, migration, replication and evacuation of so-called *adaptive applications*. An adaptive application is a service which provides the necessary information to be run and managed on a remote host, which also runs the Vagabond2 runtime. Although service adaptations are currently performed in *best-effort* manner, real-time aware service implementations are possible by using e.g. ACE [12] and the TAO ORB [7]. In contrary, a not necessarily real-time aware server like e.g. an adaptive web server might provide a Java implementation.

3 Service-Oriented Components of a Distributed Multimedia Streaming Server

Before discussing the server components of an adaptive distributed multimedia streaming server in detail, a brief connection to the ADMS runtime environment including its service-based middleware Vagabond2 is provided.

3.1 AdaptiveApplication: The Service-Oriented Component Interface of Vagabond2

The runtime environment for an adaptive multimedia streaming server could look like the one illustrated in figure 1, as proposed in [2]. It provides means for non-real-time component management, and real-time component interactions.

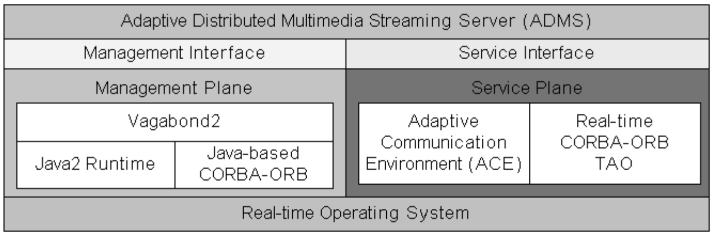


Fig. 1. The ADMS runtime environment

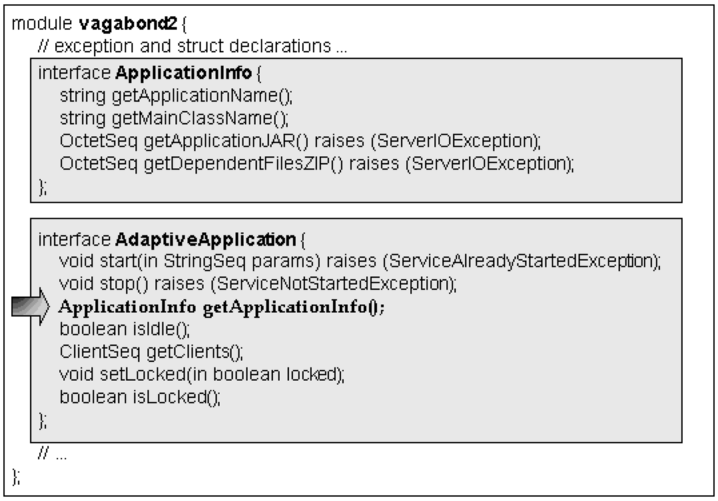
The component-based streaming server ADMS builds on top of a management plane and a service plane. The management plane interfaces to Vagabond2, which itself requires its runtime environment including a Java-based CORBA ORB for performing best-effort component adaptations. On the other hand, the service plane allows server components to directly interact by taking into account real-time constraints. Thus, the service plane is constituted by the native component implementations based on e.g. ACE and the TAO real-time CORBA ORB [7].

Vagabond2 allows to specify an adaptive service component by deriving it from the CORBA interface `AdaptiveApplication` [1,2]. Here the terms application and service are considered synonymously. The key functionality of this interface is the `getApplicationInfo()` method, which allows to dynamically request the binaries of the service component, as well as a possibly dynamic set of files, upon which the component depends on. Figure 2 illustrates this connection as an excerpt of Vagabond2's IDL specification.

The `getApplicationJAR()` method indicates that Vagabond2's implementation is Java-based, allowing for a dynamic replication/migration of the component based on byte code. As a result, each adaptive service component has to provide a Java implementation of its specification. However, using JNI, the service of the component can use a native implementation of it in order to handle real-time issues.

3.2 The Building Blocks of an ADMS

When designing an adaptive distributed streaming server, the following two guidelines have to be taken into account. First, each adaptive component must be *independent* to a certain extent, *reusable*, *adaptable*, *combinable*, and *movable*.



```
module vagabond2 {
    // exception and struct declarations ...
    interface ApplicationInfo {
        string getApplicationName();
        string getMainClassName();
        OctetSeq getApplicationJAR() raises (ServerIOException);
        OctetSeq getDependentFilesZIP() raises (ServerIOException);
    };

    interface AdaptiveApplication {
        void start(in StringSeq params) raises (ServiceAlreadyStartedException);
        void stop() raises (ServiceNotStartedException);
        ApplicationInfo getApplicationInfo();
        boolean isIdle();
        ClientSeq getClients();
        void setLocked(in boolean locked);
        boolean isLocked();
    };
    // ...
};
```

Fig. 2. Vagabond2's core interfaces for adaptive service components

And second, a component should fulfill a complete dedicated task for a certain usage scenario. E.g. during a data acquisition scenario a media stream has to be stored on a set of server nodes. To perform this operation in a distributed environment, one component is needed to receive the media stream, split the stream into smaller pieces, and distribute the pieces among a set of data nodes. The data nodes themselves are equipped with a component for storing and retrieving pieces of media data. Thus, a distribution component can be combined with a number of storage components, which all run on separate server nodes. The second guideline implies a beneficial side-effect that the number of distinct components is kept small, since the number of usage scenarios is quite limited.

Following these guidelines, four basic building blocks have been identified to constitute an ADMS: *data distributors* (DDs), *data managers* (DMs), *data collectors* (DCs), and *cluster managers* (CMs). Each of these components can be replicated or migrated on demand, and provides services to other components. Figure 3 demonstrates a sample ADMS consisting of two DDs, four DMs, two DCs, and one CM. In a typical ADMS environment there is only one CM instance, but a number of DD, DM, and DC instances, where each instance should run on a dedicated host.

3.2.1 Data Distributor: A data distributor is a service-based component that distributes media data received from a production client or a live camera to a selected couple of data managers. The unit of distribution is a so-called *stripe unit*, which either can be of constant data length (CDL), or constant time length (CTL). Like in RAID level 5 systems parity units are generated, in order to cope with data manager failures. In cases of a non-live source, the process of distribution can be driven by a MPEG-7 document describing a temporal

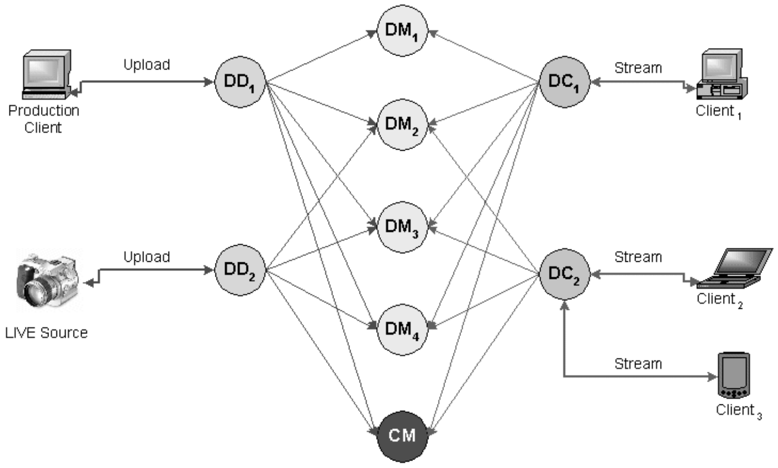


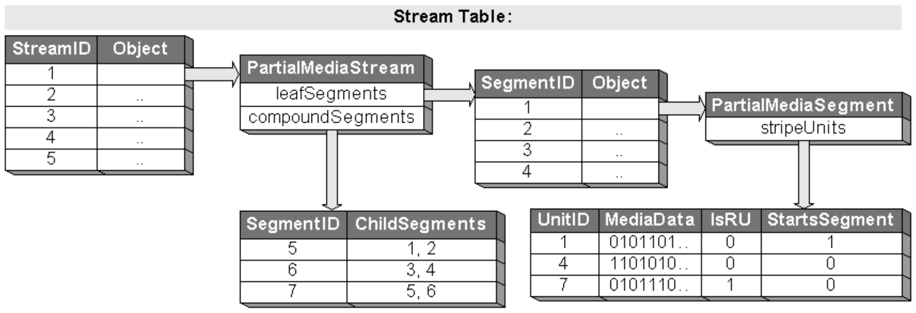
Fig. 3. Service-oriented components building an ADMS system

decomposition of the media stream. Thus, a media stream can be decomposed into a number of segments, organized in an arbitrary number of levels.

Depending on the number of data managers the media stream is striped across, the stream is either single, narrow, or wide striped. The number and location of target data managers is advised by the cluster manager component. The data distributor distributes only elementary streams, since the target system is designed for streaming scenarios based on RTP. Thus, if the media source contains a multiplexed stream, it has to be demultiplexed into elementary streams before striping. In this case, byte boundaries of elementary stream segments have to be adapted in a possibly supplied MPEG-7 descriptor.

3.2.2 Data Manager: Data managers are the key components in the ADMS architecture. A data manager provides means for efficient storage and retrieval of elementary streams or segments of them. Since one elementary stream or segment may be striped among a number of data mangers, each data manager only stores a portion of the stream or segment. Figure 4 gives an insight about how a data manger internally organizes its managed media streams. It consists of a set of partial media streams, which themselves consist of a set of leaf and compound media segments. Only leaf media segments store real media data. Stream segmentation is supported to perform more efficient media data buffering.

How a data manager is derived from Vagabond2's `AdaptiveApplication` interface is illustrated in figure 5. First, a common abstraction layer is introduced, which is valid for all four ADMS component types. It introduces the interfaces `ADMSServerApplication` and `Session`, allowing to create rate-controlled and transaction-based sessions of certain type (retrieval, acquisition, or management) with the component. Second, the bottom layer defines the interfaces and



StreamID	Object
1	
2	..
3	..
4	..
5	..

SegmentID	Object
1	
2	..
3	..
4	..

SegmentID	ChildSegments
5	1, 2
6	3, 4
7	5, 6

UnitID	MediaData	IsRU	StartsSegment
1	0101101..	0	1
4	1101010..	0	0
7	0101110..	1	0

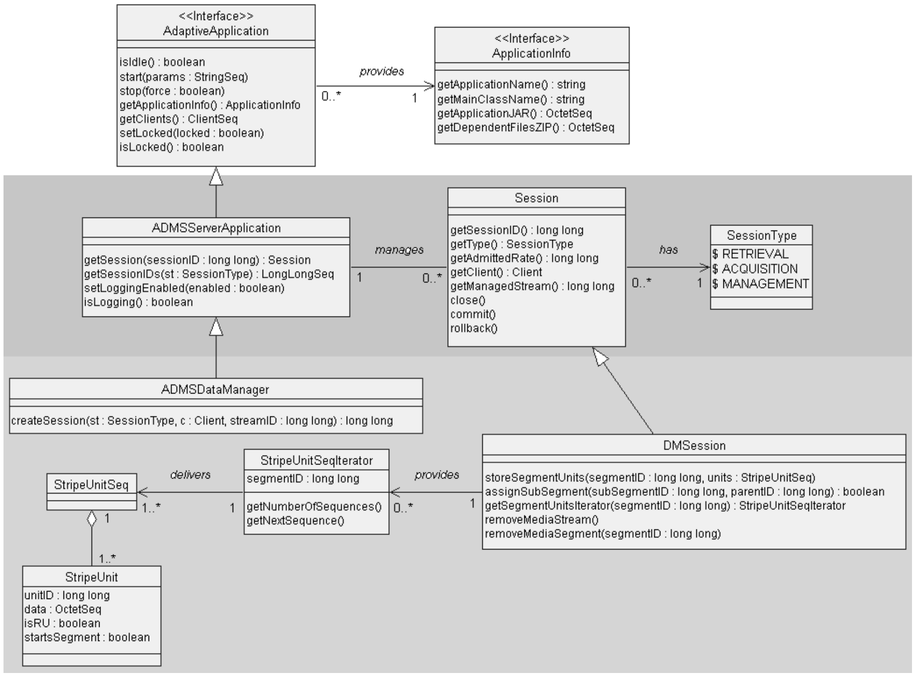
Fig. 4. Objects comprising a data manager component

structures comprising the data manager component. An `ADMSDataManager` allows only to create so-called data manager sessions (`DMSession`). Since each session is associated with exactly one elementary stream, a data manager session provides means to e.g. store stripe units for a certain stream segment, to compose a segment tree of known segments by the component, or to retrieve stripe units of a certain segment via a stripe unit iterator. Based on the sessions admitted data rate (in kbit/sec), the unit iterator allows to retrieve an according number of stripe units per second.

3.2.3 Data Collector: A data collector component performs the inverse operations of a data distributor. Its main task is to serve a client request by collecting stripe units of a certain media stream from a given set of data managers, resequencing those units, and sending the buffered stream to the client using RTP. It provides server-level fault tolerance by exploiting parity units in cases of unavailable data managers.

Since collecting stripe units from the associated data managers is always a load-generating and time-consuming task, the data collector should also include a caching component, in order to reduce startup latencies. Following Lee's framework of video distribution architectures [13], the data collector can implement the *independent-proxy* or *proxy-at-client*-model. However, there are advantages of running the data collector component as a service on a separate Vagabond2 host. First, the client does not know anything about the internal ADMS layout. Second, the collector can serve a set of clients with even unequal client capabilities by applying gateway functionalities, e.g. by transcoding the streams. This is illustrated in figure 3 for DC_2, which streams the same media stream with different qualities to a notebook and to a handheld client simultaneously. And finally, keeping the data collector component independent from the clients allows the cluster manager to select the most appropriate one for serving a certain client request.

3.2.4 Cluster Manager: Since a typical ADMS environment consists of a dynamic number of DDs, DMs, and DCs, one central component is required



<<Interface>> AdaptiveApplication
isIdle() : boolean
start(params : StringSeq)
stop(force : boolean)
getApplicationInfo() : ApplicationInfo
getClients() : ClientSeq
setLocked(locked : boolean)
isLocked() : boolean
provides
0..*
1
<<Interface>> ApplicationInfo
getApplicationName() : string
getMainClassName() : string
getApplicationJAR() : OctetSeq
getDependentFilesZIP() : OctetSeq
ADMSServerApplication
getSession(sessionID : long long) : Session
getSessionIDs(st : SessionType) : LongLongSeq
setLoggingEnabled(enabled : boolean)
isLogging() : boolean
manages
1
0..*
Session
getSessionID() : long long
getType() : SessionType
getAdmittedRate() : long long
getClient() : Client
getManagedStream() : long long
close()
commit()
rollback()
has
0..*
1
SessionType
$ RETRIEVAL
$ ACQUISITION
$ MANAGEMENT
ADMSDataManager
createSession(st : SessionType, c : Client, streamID : long long) : long long
StripeUnitSeq
delivers
1..*
1
StripeUnitSeqIterator
segmentID : long long
getNumberOfSequences()
getNextSequence()
provides
0..*
1
DMSession
storeSegmentUnits(segmentID : long long, units : StripeUnitSeq)
assignSubSegment(subSegmentID : long long, parentID : long long) : boolean
getSegmentUnitsIterator(segmentID : long long) : StripeUnitSeqIterator
removeMediaStream()
removeMediaSegment(segmentID : long long)
1
1..*
StripeUnit
unitID : long long
data : OctetSeq
isRU : boolean
startsSegment : boolean

Fig. 5. The data manager component as special adaptive application of Vagabond2

for managing these component instances and compositing the virtual server. The cluster manager component does this and furthermore implements a directory service keeping track which data mangers store data of which elementary streams, and which data collectors keep which elementary streams in their cache. For data acquisition it advises data managers for stream distribution by applying load-balancing strategies for network and host resources. For data retrieval it originally handles client requests based on RTSP, and delegates the request to appropriate data collectors, if possible. These decisions are based on server loads, network loads, client vicinity to data collectors, and client terminal capabilities.

4 Dependence Management between ADMS Components

Similar to the dependence management of component-based applications in operating systems suggested in [14], a dynamic dependence management between ADMS components is necessary. Despite the obvious dependency of distributors and collectors from the cluster manager, a collector may be faced with dynamic, stream-based dependencies from data managers. In particular, a data collector dc depends on a data manager dm regarding a media stream m, iff dc does not have cached m, and dm stores at least a portion of m (m_p). Speaking in terms of a component configurator as presented in [14], the hook from dc to dm is dynamic, since it is removed, if either dc caches m, or dm is replaced by another data manager dm_r. The replacement itself can either be accomplished by replicating the partial media stream m_p to an existing data manager, or to a newly created data manager, or even by replicating the whole data manager dm to a new host.

Different dependence configurations may result in considerably different performance behaviors when serving a client request. This is especially the case, if the locations of a data collector's hooks are wide spread. Performance evaluations in the ADMS testbed, consisting of servers in two LANs interconnected by the internet with widely distributed data managers, have shown similar results. For detailed results the interested reader is referred to [1]. It has to be noted, that the dependence management is accomplished by the cluster manager. The CM is also responsible for taking care of optimal dependencies between data collectors and data managers. Thus, if it recognizes increased request denials due to bad collector hooks, it has to reconfigure the server layout by replicating/migrating media streams and/or ADMS server components.

5 Conclusion and Future Work

The major contribution of this paper lies in exploring and designing the building blocks of an adaptive distributed multimedia streaming server. Current distributed multimedia servers have deficits in adapting themselves to changing client demands due to their monolithic implementations. Four basic components have been identified that allow to compose a distributed streaming server, which is able to adapt itself in reaction to varying client demands. Furthermore, it is shown how these components are built on top of Vagabond2, which serves as the middleware for component management and adaptation. The contribution is concluded by pointing out the explicit inter-component dependencies, which have to be considered during server adaptations.

A summarization regarding the common values of most important properties of ADMS components is given in Table 1. While a data manger e.g. is highly adaptable due to the coding characteristics and the amount of the media data it stores, a cluster manager is much less adaptable. On the other hand, a cluster manager is much more mobile than a data manager, since it depends on orders of magnitude less data.

Table 1. Values for most important properties of ADMS components

Component	Dependencies	Adaptability	Combinability	Mobility
CM	none	low	DD, DC	high
DD	CM, DM	medium	CM, DM	high
DM	none	high	DD, DC	low
DC	CM, DM	medium	CM, DM	medium

Currently a resource broker component is being developed which allows to monitor and manage network and host resources between all server hosts. This component will be used by the cluster manager to decide whether a client request will be admitted or not. Future plans are to integrate a native implementation of a MPEG-4 proxy server as a data collector component in the ADMS system. Since this proxy server comprises gateway functionality, the data collector will even be able to adapt media streams to different qualities. This approach opens a new research area concerning combined adaptation possibilities regarding server adaptations and media stream adaptations.

References

1. Goldschmidt, B., Tusch, R., Böszörményi, L.: A Mobile Agent-based Infrastructure for an Adaptive Multimedia Server. Parallel and Distributed Computing Practices, Special issue on DAPSYS 2002 (2003) To appear. Papers is also available as technical report TR/ITEC/03/2.05.
2. Tusch, R.: AMS: An Adaptive Multimedia Server Architecture. Technical Report TR/ITEC/02/2.06, Institute of Information Technology, Klagenfurt University (2002)
3. Object Management Group: Common Object Request Broker Architecture: Core Specification. 3.0.2 edn. (2002) http://www.omg.org/cgi-bin/doc?formal/02-12-02.pdf.
4. Microsoft: DCOM Technical Overview. (1996) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp.
5. Sun Microsystems: Enterprise JavaBeans Specification. 2.1 edn. (2002) http://java.sun.com/products/ejb/.
6. Object Management Group: Real-Time CORBA Specification. 1.1 edn. (2002) http://www.omg.org/cgi-bin/doc?formal/02-08-02.pdf.
7. Schmidt, D.C., Levine, D.L., Mungee, S.: The Design of the TAO Real-Time Object Request Broker. Computer Communications, Elsivier Science **21** (1998)
8. Object Management Group: Audio/Video Stream Specification. 1.0 edn. (1998) http://www.omg.org/cgi-bin/doc?formal/00-01-03.pdf.
9. Mungee, S., Surendran, N., Krishnamurthy, Y., Schmidt, D.C.: The Design and Performance of a CORBA Audio/Video Streaming Service. In: Design and Management of Multimedia Information Systems: Opportunities and Challenges. Idea Group Publishing, Hershey, USA (2001)
10. Waldo, J.: The Jini Architecture for Network-centric Computing. Communications of the ACM **42** (1999) 76–82
11. Friedman, R., Biham, E., Itzkovitz, A., Schuster, A.: Symphony: An Infrastructure for Managing Virtual Servers. Cluster Computing **4** (2001) 221–233
12. Schmidt, D.C.: The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. Washington University. (1994)
13. Lee, J.Y.: Parallel Video Servers: A Tutorial. IEEE Multimedia **5** (1998) 20–28
14. Kon, F., Campbell, R.H.: Dependence Management in Component-Based Distributed Systems. IEEE Concurrency **8** (2000) 26–36

A Layered Type System for Re-entrance Control

Simon Kent

Centre for Information Technology Innovation
Queensland University of Technology
Level 4, 126 Margaret St.
Brisbane, QLD, 4000, Australia
s.kent@qut.edu.au
http://www.citi.qut.edu.au

Abstract. Re-entrance can cause errors in stateful objects by exposing transient states; however, re-entrance can only be constrained programmatically in most modern object-oriented languages. Re-entrance is a global system behaviour that is of particular concern in component-based applications, which are, by definition, incomplete. This paper describes programming language constructs that constrain re-entrance by bounding control flow through stateful objects. Interface types are extended with object state information while component types (modules) are extended with a layering over those types. The type system uses a combination of dynamic and static checking to bound control flow through stateful objects, while a composition check is used to ensure the combined type layering of a program is consistent. Our extensions restrict unchecked re-entrance while allowing a programmer to design for specific cases of re-entrance where necessary.

1 Introduction

Re-entrance can be a source of subtle errors in compositions of stateful objects as it can expose intermediate states during complex interactions. Re-entrance can only be constrained programmatically in most modern O-O programming languages, however, and traditional pre- and post-conditions can easily be broken by re-entrance if not carefully designed. This is of particular concern when O-O languages are used to implement software components as re-entrance across component boundaries can cause errors that are difficult, if not impossible, to debug [1]. In this paper, we show how re-entrance can be constrained by adding abstract states to types, and placing bounds on how those states may be transformed.

Re-entrance occurs when an object's method is invoked while another of the object's methods is still executing. We note that:

1. Re-entrance is a temporal behaviour of stateful objects. More specifically, re-entrance occurs when stateful objects are already undergoing a state transformation.

2. Re-entrance is a global system behaviour that occurs as the result of interactions between objects.

Our approach for constraining re-entrance relies on extended types that model the two above properties:

1. We extend object types (called *negotiable interfaces*) with protocols that specify legal state transformations for an object, including recursive or re-entrant transformations.

2. We extend component types (modules) with *type layers* that specify legal sets of state transformations for interacting objects.

The type system uses a combination of static and dynamic checking to ensure a program transforms objects and sets of stateful objects according to the protocol types and type layers. Dynamic checking, called *negotiation*, is used as kind of runtime type-test and guard on the state of an object and program, while static checking is used to ensure that an object is transformed according to its protocol once the object's state is known. The use of dynamic checking lets us avoid problems with state-space explosion and aliasing that a purely static scheme would need to address.

Type layer declarations also provide us with a simple composition check for modules (our units of composition) that ensures no set of modules have conflicting type layering requirements.

In the following section we informally describe *negotiable interfaces* and show how they may be used to design for re-entrance. Section 3 introduces type layers and demonstrates how they are combined with negotiation to bound object interactions. Section 4 addresses related work, while future directions and conclusions are discussed in Sect. 5.

We demonstrate our language extensions using a Pascal-like syntax.

2 Negotiable Interfaces

Negotiable interfaces are interface types extended with a behaviour protocol that specifies:

– The abstract states that an object may have.
– The abstract state transformations that the object may undergo.

Each method of a negotiable type has a state transformation, or *behaviour*, that specifies the required pre-state and final post-state of the method, as well as an optional *re-entrant behaviour*.

Informally, each object of a negotiable type has a factorable, abstract state associated with it. To invoke a state transforming method, a client must first have a *stateful* reference to an object; a stateful reference carries part of an object's abstract state and may be obtained by *negotiating* with an object. Stateful references may also be passed as parameters to other methods or routines.

A more detailed discussion of basic negotiable interfaces that addresses static checking and semantics is presented in [2].

```
1: TYPE
2:  IFrame* = POINTER TO INTERFACE RECORD
3:              STATE Min,Max,Item,Position;
4:              INIT  [Min];
5:            END;
6:
7:  PROCEDURE (this:IFrame)
8:    Maximize*() :: [Min]->[Max];
9:
10: PROCEDURE (this:IFrame)
11:   PositionItem*(i:IFrameItem) :: [Position]->[Position];
12:
13: PROCEDURE (this:IFrame :: [Position] -> [Position])
14:   Host*(i:IFrameItem) :: [] -> [Item];
15:   ...
```

Fig. 1. A declaration of a negotiable interface

2.1 Negotiable Interface Types

Figure 1 demonstrates the syntax for declaring negotiable interface types. The `STATE` declaration on line three declares the state descriptors, or *tokens*, of the interface. The `INIT` declaration on line four declares the initial abstract state (a **bag** of tokens) for a newly created object of this type.

Figure 1 also demonstrates negotiable method declarations. On line eight the method `IFrame.Maximize` is declared to transform the state of an `IFrame` object from `[Min]` to `[Max]`. Lines 13 and 14 demonstrate a re-entrant method declaration: the method `IFrame.Host` adds an `Item` to the state of an `IFrame` object and declares a re-entrant behaviour, `[Position] -> [Position]`. This indicates that during execution of `IFrame.Host` the object should expect a sequence of method invocations that satisfies the behaviour `[Position] -> [Position]` (i.e. zero or more calls to `IFrame.PositionItem`).

2.2 Negotiable Interfaces and Re-entrant Methods

Figure 2 shows a simple example of a re-entrant state transformation.

On line 20 is a routine, `AddStandardMenu`, that attempts to add two menus to an `IFrame` object. `AddStandardMenu` negotiates for a stateful reference on line 23, and promises to add two `Items` to the state of the `IFrame`. If the negotiation is successful, the routine calls the re-entrant method `IFrame.Host` twice to host the two menus in the frame.

Line 12 shows an implementation of `IFrame.Host`. Apart from transforming the abstract state of the object by adding an `Item`, the method also has a re-entrant behaviour, `[Position] -> [Position]`. This means the receiver parameter, `this`, is a stateful reference which permits the invocation of a sequence of recursive or re-entrant method calls that satisfies the re-entrant behaviour.

```
1: TYPE
2:   Frame* = POINTER TO RECORD (ANYREC + IFrame) ... END;
3:
4:  PROCEDURE (this:Menu.IFrameItem)
5:    Layout*(parent:IFrame :: [Position]->[Position]);
6:  BEGIN
7:    ...
8:    parent.PositionItem(this); (* parent :: [Position] -> [Position] *)
9:    ...
10: END Host;
11:
12: PROCEDURE (this:Frame.IFrame :: [Position]->[Position])
13:   Host*(i:IFrameItem) :: []->[Item];
14: BEGIN
15:   ...
16:   i.Layout(this);            (* this :: [Position] -> [Position] *)
17:   ...
18: END Host;
19:
20: PROCEDURE AddStandardMenu(frame : IFrame);
21: BEGIN
22:   ...
23:   USE frame :: [] -> [Item,Item] DO
24:     ...
25:     frame.Host(fileSubMenu); (* frame :: [] -> [Item] *)
26:     frame.Host(helpSubMenu); (* frame :: [Item] -> [Item,Item] *)
27:     ...
28:   ELSE
29:     ...
30:   END;
31:   ...
32: END AddStandardMenu;
```

Fig. 2. A re-entrant state transformation

In the Frame implementation of IFrame.Host, the stateful reference is passed to IFrameItem.Layout as a parameter (line 16), allowing the IFrameItem to re-enter the Frame (line eight) to get its position while it is laying out.

In summary, re-entrant behaviours allow a programmer to:

– Specify methods that are recursive or re-entrant.
– Specify methods that re-enter an object, for each re-entrant method.
– Identify out-calls that will re-enter an object (by checking if the out-calls transforms the state of "this", as on line 16 for example).

These properties allow a programmer to design for specific cases of re-entrance, by reducing the number of methods that may be re-entered and the number of out-calls at which a programmer needs to ensure object consistency.

The next section describes how type layers are used to restrict re-entrance further.

3 Type Layers

Type layers are an extension to component types (module signatures) that specify how sets of stateful objects may be transformed in an interaction. Informally, type layers constrain object interactions by requiring that during any transformation of a stateful object of some type, say `IFoo`, a program can only initiate new state transformations in type layers "below" `IFoo`. This allows a programmer to specify an abstract architectural control flow through a program's stateful objects.

Figure 3 demonstrates our syntax for type layer declarations. Line five shows an individual layer declaration, which specifies that `IScrollBar` states are lower than `ITextBox` states, while lines six and seven demonstrate syntactic sugar for multiple layer declarations.

```
1: MODULE Widgets;
2: IMPORT Logic;
3:
4: LAYER
5:    IScrollBar < ITextBox,
6:    {ITextBox,IMenu} < IFrame,
7:    Logic.* < Widgets.*;
```

Fig. 3. Type layer declaration

Figure 4 illustrates the type layering given in figure 3. A window is composed of a window frame, a menu, and an edit box which in turn has scroll bars. A program's control flow (the solid arrow) may only negotiate downstream according to the layer specification (the dashed arrows). A program that has entered an edit box state may not attempt to re-enter the frame state, but may negotiate down to the scroll bars for example. A program may re-enter a certain state but only if it is passed explicit permission in the form of a stateful parameter as described in Sect. 2.2.

Type layer declarations are used to create a strict partial ordering of negotiable types, separate from the inheritance hierarchy or the import order. Instead, the type layer strict p.o. determines which states may be negotiated with at a given point in a program. To bound control flow through stateful objects, we define *program state* as the type of the most recent ongoing state transformation, and require that a program can only successfully negotiate with types below the program state with respect to the type layer strict p.o.

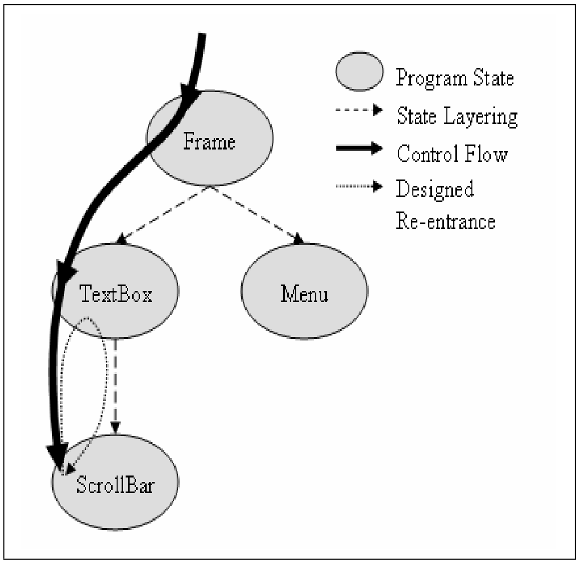


Fig. 4. Type layer specification with hypothetical control flow

3.1 Composition Checking

A program's type layer specification is the combination of type layer declarations of the program's modules. Clearly, it is possible for modules to have conflicting type layer requirements and this provides us with a simple composition check for a program. Intuitively, if two or more modules have conflicting type layer declarations, then in combination those modules may operate on stateful objects in a cyclic (possibly re-entrant) manner.

We provide a semi-formal description of the composition check in figure 5. We use two environments for composition checking: $\prec$, a relationship on negotiable interfaces, and M, a mapping from identifiers to modules.

$$\begin{aligned} \prec &: Interface \longleftrightarrow Interface \\ M &: Ident \mapsto Module \end{aligned}$$

We also define $\preceq$ to be the transitive, reflexive closure of $\prec$, and a helper function, *modulename*, that extracts the module identifier from a given module.

$$\preceq \quad = \prec^{*}$$
$$modulename(\texttt{MODULE}\ id\ \texttt{IMPORT}\ \_\ \texttt{LAYER}\ \_) = id$$

The composition check is very simple. If a program checks, then it can be proven from the rules that $\preceq$ is a partial order, and that $\prec^{+}$, the transitive closure of $\prec$, is a strict partial order. This means there are no cycles in the type layer specification, and all modules agree on how sets of stateful objects can be transformed. If the program does not check, then the layer declarations contain

Abstract syntax

$$\begin{array}{lll} Module & ::= & \texttt{MODULE}\ Id\ \texttt{IMPORT}\ Id^*\ Layers \\ Layers & ::= & \texttt{LAYER}\ Layer^* \\ Layer & ::= & Interface < Interface \end{array}$$

Type rules

Layer

$$\frac{Interface_2 \not\prec Interface_1}{\prec\ \vdash Interface_1 < Interface_2 :\ \ \prec \cup \{(Interface_1, Interface_2)\}}$$

Layers

$$\frac{\prec_{i-1} \vdash layer_i : \prec_i, i = 1, \ldots, n}{\prec_0 \vdash \texttt{LAYER}\ layer_1, \ldots, layer_n : \prec_n}$$

Module

$$\frac{\begin{array}{c} M, \prec_{i-1} \vdash M(m_i) : \prec_i, i = 1, \ldots, n \\ \prec_n \vdash layers : \prec' \end{array}}{M, \prec_0 \vdash \texttt{MODULE}\ \_\ \texttt{IMPORT}\ m_1, \ldots, m_n\ layers : \prec'}$$

Program

$$\frac{\begin{array}{c} M = \{modulename(m_i) \mapsto m_i\}_{i=1,\ldots,n} \\ M, \{\} \vdash M(mid_{MAIN}) : \prec \end{array}}{\vdash mid_{MAIN}}$$

Fig. 5. Type layer composition checking

one or more cycles, and the combined type layering cannot prevent the program from manipulating stateful objects in cyclic (possibly re-entrant) interactions.

3.2 Extensible Systems

The composition checking system presented above statically checks the unity of a program's modules, but the check can be extended to enable extensible systems in which components (modules or compositions of modules) are loaded dynamically.

At the very least, when a component is loaded we can update a program's type layering dynamically and prevent loading of components that would introduce cycles into the layering. This may be expensive for large systems, however, and we can place other conditions on dynamically loaded components that ensure such components can be loaded without introducing cycles into the layering. We omit formal details for the sake of brevity but posit that independently developed components satisfying both of the following properties can be dynamically loaded without violating type layer constraints:

– The component does not strengthen the original program layering.
– The component does not layer types that have been loaded by other components.

3.3 Semantics

Figure 6 presents a simplified operational semantics that illustrates how state transformations are bounded by type layers. The semantics steps a program consisting of a command and a store; the abstract syntax of the commands affected by type layers is also given in figure 6, while we do not specify the form or content of the store but note that it may change when stepping commands for which rules are omitted.

The environment of an executing program is a type layering, $\prec$, obtained from the composition checking rules above, and the program state, which is a value from the type layering. We also define $\prec^+$, the transitive closure of $\prec$ (a strict p.o.). and *typeof*, a helper function that obtains the static type of a reference.

For brevity, we omit the details of negotiation for object state and instead use the informal statement, *object*(*id*) *satisfies* b, to indicate that the object referred to by *id* is in the state requested by behaviour b.

Abstract syntax

Command ::= `USE` *Ident* :: *Behaviour* *Command* `ELSE` *Command* | `SKIP` | ...

Operational Semantics

Negotiation (**success**)

$$\frac{\begin{array}{c} \mathit{typeof}(\mathit{id}) \prec^+ I \land \mathit{object}(\mathit{id})\ \mathit{satisfies}\ b \\ \prec, \mathit{typeof}(\mathit{id}) \ \vdash c_1\{S\} \longrightarrow \mathtt{SKIP}\{S'\} \end{array}}{\prec, I \ \vdash \mathtt{USE}\ \mathit{id} :: b\ c_1\ \mathtt{ELSE}\ c_2\{S\} \longrightarrow \mathtt{SKIP}\{S'\}}$$

Negotiation (**fail**)

$$\frac{\begin{array}{c} \neg(\mathit{typeof}(\mathit{id}) \prec^+ I \land \mathit{object}(\mathit{id})\ \mathit{satisfies}\ b) \\ \prec, I \ \vdash c_2\{S\} \longrightarrow \mathtt{SKIP}\{S'\} \end{array}}{\prec, I \ \vdash \mathtt{USE}\ \mathit{id} :: b\ c_1\ \mathtt{ELSE}\ c_2\{S\} \longrightarrow \mathtt{SKIP}\{S'\}}$$

Fig. 6. Semantics of negotiation with respect to type layers.

Since $\prec^+$ is a strict partial order, a program cannot negotiate with a reference of the same type as the program state, nor can it negotiate with a reference of a type above the program state w.r.t. the type layering. In this way, type layering prevents a program from re-entering, or re-negotiating, for states it is already manipulating. Only by designing for re-entrance by using recursive behaviour descriptions, or by implementing types lower down the type layering, can an object be re-entered. This means the programmer no longer has to presuppose all possible re-entrance scenarios, but only be concerned with specific cases.

4 Related Work

The concept of associating an abstract state to objects in order to describe changing method availability is not a new one. In particular, many concurrent object-oriented languages and type systems for active objects use a concept of object state to help determine method availability. More recently, protocol types have also been applied to interfaces for component programming.

Many concurrent O-O languages that use abstract state to determine method availability have “active” objects such as those in Actor languages [3]. Different schemes for using abstract state for concurrency control, such as path expressions in PROCOL [4] and BCOOPL [5], behaviour sets in ACT++ [6] and guards in Guide [7], result in differing levels of expressiveness for determining method availability. As these languages deal mainly with active objects the focus is placed upon the ability of objects to handle specific orderings of messages. Specification and prevention of re-entrance is not a concern of such languages so expressing intermediate state during method invocation is generally not considered.

Type systems for active objects such as in [8] [9] [10], [11], and [12], annotate the methods of interfaces or classes using some process formalism to describe the changing availability of services during execution. As the focus of such type systems is to type behaviour patterns in languages with active objects, describing re-entrance and intermediate state is again not a primary focus.

The type system presented and developed primarily by Puntigam in [10,11] and Puntigam and Peter in [12] has a similar scheme for decorating interfaces with (extended) bags of tokens and assigning behaviours to methods. Our proposal was directly influenced by this work as bags are factorable and may be split over references. The object model and objectives of the type system proposed here are different from that of Puntigam and Peter. They consider active objects that communicate via asynchronous message passing with unbounded buffers - once again re-entrance is not a concern.

Finally, the type system described by Reussner in [13,14] extends Java interfaces with augmented finite state automata. Reussner combines automata describing legal ordering of calls with automata that describe the services a component uses. The combined automata can be checked with behaviour of other components at composition time to accept or reject legal compositions. Reussner’s type system focuses upon adaptation of components and composition level checking, whereas our type system is aimed at component construction and is accompanied with the runtime mechanism of negotiation.

5 Conclusions and Further Work

Negotiable interfaces allow a programmer to bound the behaviour of stateful objects, while type layers allow a programmer to bound interactions of sets of stateful objects. In particular, these extensions restrict unchecked re-entrance but can be used to express specific cases of re-entrance when necessary. Type layering also provides an enhanced notion of composition that checks if two components have compatible control flow requirements across stateful objects.

We aim to extend negotiable interfaces to support re-use through object composition at the type level by allowing stateful objects to be "plugged" together using abstract states. We are also investigating an extension to the type layering system that allows pseudo-cycles in the layers, so that we can bound interactions between hierarchical compositions of objects of the same type.

The language extensions described in this paper have been completely implemented in an experimental version of `gpcp` [15], a Component Pascal compiler that targets the .NET CLR and the JVM.

References

1. Szyperski, C.: Component Software, Beyond Object–Oriented Programming. 1 edn. Addison–Wesley, Harlow, Essex (1998)
2. Kent, S., Ho-Stuart, C., Roe, P.: Negotiable interfaces for components. Journal of Object Technology, Special Issue: TOOLS USA 2002 proceedings **1** (2002) 249–265
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. Series in Artificial Intelligence. MIT Press, Cambridge, MA (1986)
4. van den Bos, J., Laffra, C.: PROCOL: A concurrent object-language with protocols, delegation and persistence. Acta Informatica **28** (1991) 511–538
5. de Bruin, H.: BCOOPL: Basic concurrent object-oriented programming language. Software - Practice and Experience (SPE) **30** (2000) 849–894
6. Kafura, D.G., Lee, K.H.: ACT++: Building a concurrent c++ with actors. Journal of Object-Oriented Programming **3** (1990)
7. Balter, R., Lacourte, S., Riveill, M.: The Guide language. The Computer Journal **37** (1994) 519–530
8. Nierstrasz, O.: Regular types for active objects. In Nierstrasz, O., Tsichritzis, D., eds.: Object-Oriented Software Composition. Prentice Hall (1995) 99–121
9. Wehrheim, H.: Subtyping patterns for active objects. In: Proceedings 8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung): Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme, Muenster (2000)
10. Puntigam, F.: Types for active objects based on trace semantics. In: 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), Paris, France (1996)
11. Puntigam, F.: Coordination requirements expressed in types for active objects. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97). Springer–Verlag, Jyväaskylä, Finland (1997)
12. Puntigam, F., Peter, C.: Changeable interfaces and promised messages for concurrent components. In: ACM Symposium on Applied Computing (SAC'99), San Antonio, Texas (1999)
13. Reussner, R.: Formal foundations of dynamic types for software components. Technical Report 08/2000, Department of Informatics, Universität Karlsruhe, Department of Informatics (2000)
14. Reussner, R.: Enhanced component interfaces to support dynamic adaptation and extension. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS–34). (2001)
15. gpcp: Gardens point component pascal. http://www.fit.qut.edu.au/CompSci/PLAS/ComponentPascal/ (2001)

A Practical Approach to Software Portability Based on Strong Typing and Architectural Stratification

Andrey Koltashev

The Reshetnev NPO Pricladnoi Mekhanici
Onboard Software Department
662972 Lenina Street, 52, Zheleznogorsk, Krasnoyarsk region, Russia
216kaa@npopm.ru
http://www.npopm.ru

Abstract. This paper describes an approach to porting onboard software for communication satellites to new platforms that use various onboard computers and devices. The approach relies on the strong typing and separate compilation of Modula-2 and allows to port up to 80% of the application software.

1 Introduction

The industry of communication satellites in Russia has to use various computing hardware in functionally equivalent satellites due to economical, technical (weight, containment, etc.) and political (export restrictions) considerations. So there arises the problem of porting the onboard software (OSW) to various computing platforms.

The problem is being resolved by the Reshetnev NPO in collaboration with the Ershov Institute of Informatics Systems and the Excelsior, Ltd. (formerly XDS), via an approach based on architectural stratification and interface standardization both for the onboard software and the development environment [2,3,4].

First of all, a stratification of the OSW was performed which allowed us to define and implement an OSW Abstraction layer that provides a standard platform-independent API (types and procedures) to application software (i.e. the software of satellite subsystems that solves functional problems and amounts to up to 80 % of the entire OSW). Portability of the application software to new hardware is ensured by the software development system whereas the use of a new operating system and drivers, by reprogramming the implementation modules for the OSW Abstraction layer.

Next, a stratification of the cross-programming system (CPS) was made which allowed us to define platform-independent and programming-language-oriented user interfaces: both for programming and testing/debugging needs, and the CPS Abstraction layer that provides a standard architecture-independent API (types and procedures) that isolates architecture–dependent parts of the onboard computers: CPS components implement code generation and instruction set simulator.

A successful implementation and an efficient use of both the OSW and CPS Abstraction layers proved to be possible to a great extent due to the strong typing and separate compilation properties of Modula-2.

Moreover, the use of a highly structured language provided an additional benefit, namely, a possibility to measure static and dynamic software component characteristics and to calculate criterions of software testing completeness for quality assurance.

2 Onboard Software Stratification

A canonical three-layer structure of the OSW was determined. The third layer — the application software — becomes platform-independent because its implementation only uses interfaces exported by the lower, second layer called the Abstraction layer. The first layer is the employed operating system, i.e. a real-time kernel and a set of drivers.

The canonical structure of the OSW Abstraction layer for the given problem area was determined by the analysis of functionality of several generations of satellites.

Three components are specified in the Abstraction layer:

– standard interfaces for a canonical set of abstract real-time kernel calls;
– standard interfaces for a canonical set of abstract onboard devices;
– standard types for a canonical abstract data set used to control subsystems of the satellite in various regimes.

The main features of Modula-2 — the strong typing and separate compilation — gave a possibility to implement the Abstraction layer as a set of Modula-2 libraries providing a complete and platform-independent API. It proved possible to define the data types independently of addressability, endian-ness, word size, etc., of various onboard computers.

The strong typing, e.g. enumeration types, provided a required level of abstraction for data types which is both maximally close to the problem field and also platform-independent to a maximal degree.

The separate compilation feature allowed us to standardize and freeze the API using definition modules, and made it possible to reuse all the functionally equivalent application software without any changes in the source code, which greatly simplified configuration management.

3 A Modula-2 Cross-Programming System Adaptable to Different Target Onboard Computers

We developed a cross-programming system (CPS) based on Modula-2, which implements the code generation for some base computer implemented via an instruction set emulator. In this CPS, there were implemented platform-independent programming-language-oriented user interfaces as well as the CPS Abstraction layer that provides a standard architecture-independent interface with parts that depend on the architecture of the target onboard computer: the CPS components that implement code generation and the instruction set simulator.

Adaptation for a new embedded computing system is performed by making use of commercial C compilers that are always available. In this case the code generation is performed transparently in two steps: conversion of the Modula-2 code into ANSI C by a converter within the CPS and compilation of the resulting C code by commercial compilers. The CPS ensures that all debugging information is provided at the level and in terms of Modula-2.

The assembly of the project into executable code is done via a standard CPS shell with the use of the necessary components of the commercial software development system. An important fact is that the C compiler used in the CPS is entirely hidden from application software programmers. Another important fact is that application programmers can notice whether a switch to a different target computer (another command set interpreter) has occurred only through changes in the program operating characteristics as measured by the CPS, because all the user interfaces stay exactly the same.

The CPS also includes testing and debugging facilities that use platform-independent testing languages. The platform-independent dialogue and batch testing languages allow programming, executing and documenting test procedures in the platform-independent terms of the programming language. The batch testing language allows one to reuse the test procedures with another platform.

The CPS runs efficiently under MS Windows NT/2000 on Intel Pentium chips.

Such an approach was influenced by the XDS programming environment [5], and the XDS system (which contains both Modula-2 and Oberon-2 compilers) was used as a programming environment for the CPS development and maintenance.

So, the architectural stratification of the OSW and CPS and the standardization of the two Abstraction layers ensures an easy adapation of the CPS to a new target computing platform.

4 An Additional Benefit of Using a Highly Structured Language

Our CPS allows one to obtain a complete set of measurements for the program unit being developed (including the size of stacks, the execution time, etc.) in order to evaluate some measures of source code quality and to calculate the C1 (all branches) and C (all decisions) criterions of unit testing completeness.

This is only possible with a highly structured language, so this quality allows one to implement in the CPS a complete check of the program execution process during the testing procedure and to make the testing procedure (especially under batch testing) independent of local changes in the source code of the program being tested.

5 Conclusion

The described technology of the OSW development results not only in a high level of reuse and portability of the OSW, but also a possibility to reuse all testing procedures for regressive tests of the OSW.

This is true not only for the OSW development, but also for the CPS. Moreover, it is possible to start the OSW development not waiting for adaptation of the CPS to a new target onboard computer by using the CPS available for the base onboard computer.

The measurement tools of the CPS that are based on the properties of the programming language allow one to achieve the high level of quality of the OSW components required for the spaceflight applications.

The standard and stable user interfaces of both the CPS and the Abstraction layer greatly simplify the maintenance of OSW by facilitating experience accumulation and transfer.

The technology successfully worked in the most complicated situations. For instance, the OSW developed for a reference 32-bit little-endian byte-addressable CPU and for a reference operating system, was ported to a 16-bit big-endian word-addressable CPU running under the operating system of the computer supplier.

Acknowledgments. I would like to thank my collaborators of many years professor I.V. Pottosin, E.P. Kuznetsov, A.D. Khapugin and A.V. Yeryomin without whose contributions the implementation of the described system would have been impossible.

References

1. Вирт Н. Программирование на языке Модула-2 / Пер. с. англ. - М.: Мир, 1987.
2. Pottosin I. V. SOCRAT: Programming environments for Embedded Systems. – Novosibirsk., 1992 – 20 p. - (Prepr. / Siberian Division of the Russian Academy of Science, Institute of Informatics Systems, № 11)
3. Koltashev A.A. A programming environment for embedded computers: requirements and trends // Programming environments: methods and tools / Ed. by I.V. Pottosin. – Siberian Division of the Russian Academy of Science, Institute of Informatics Systems, Novosibirsk, 1992.
4. Koltashev A.A. Computer independent technology of onboard software development. // The International scientific – practical Conference "SASS-2001" / Siberian Aerospace Academy. - Part II. - Krasnoyarsk, 2001.
5. Native XDS-x86 (User's guide) //The XDS product family / XDS Ltd.- 1997.

Object Life-Cycle Management in a Highly Flexible Middleware System

Karl Blümlinger, Christof Dallermassl, Heimo Haub, and Philipp Zambelli

Institute for Information Processing and Computer Supported New Media (IICM)
Graz University of Technology, Austria
Inffeldgasse 16c 8010 Graz, Austria
{kbluem,cdaller,hhaub,pzamb}@iicm.edu

Abstract. This paper describes the object life-cycle management in the Dinopolis middleware system. Advanced object composition is used to support adaptability and extensibility of the running system. Such a high degree of flexibility requires mechanisms to keep objects and the entire system in a consistent state. We show that a fine-grained definition of the object life-cycle helps controlling objects in a highly flexible fashion. Moreover, we discuss how the life-cycle of composed objects can be exploited to model different dynamic scenarios found in distributed, dynamically changing environments.

1 Introduction

Two important capabilities of modern distributed systems are adaptability and extensibility of the running system. In order to be able to react on a dynamically changing, heterogenous environment it has to be possible to reconfigure the system at runtime. Design for late composition [1] and loose coupling has to be applied since developers are confronted with the problem that the concrete environment in which their software is deployed is first known at runtime. Object composition, if not done systematically, results in software which is difficult to manage and hard to understand. Composed objects have to be built up methodically and the system has to keep them in a consistent state. A well-defined life-cycle and a systematic way for restructuring composite objects is necessary to ensure integrity of the whole system. In this paper, we describe how the life-cycle of objects is controlled in *Dinopolis*. The Dinopolis middleware system is a highly modularized, distributed object system providing seamless integration and virtualization for arbitrary external systems. In this article, we focus on one of the central parts of the Dinopolis system: The *object management module* is responsible for the internal structure of *Dinopolis objects* and controls their life-cycle. The object management module is part of the kernel of Dinopolis and all calls which reach the kernel are already security checked in the *Kernel Access Layer* which is the layer between kernel space and user space. The Dinopolis kernel itself consists of several kernel-modules. For a description of the overall system architecture and all sub-modules please refer to [2,3]. A long version of this paper is available at http://www.dinopolis.org/publications/.

2 The Dinopolis Object

Dinopolis objects are the entities which are requested by applications. They are long-living objects (see [4]) and their life-cycle is not limited to their existence in memory. They are associated with a robust, globally unique handle which is kept stable even if they are removed from memory or moved around in the system (see [2,5]). Object-mobility is one of the key features of Dinopolis objects, since they encapsulate medium-sized entities as, for example, an XML file stored in a file-system. This points out another strong characteristic of Dinopolis: The system provides a possibility to integrate arbitrary *external systems* [2] like file-systems or databases which can be combined to so-called *virtual systems.* A virtual system then provides the superset of functionality of a number of external systems. Although some parts of the Dinopolis object can be considered to be part of a static interface, it is absolutely impossible to define statically *how* the implementation looks like. One of the few predictions to be made is that the implementations will come from different modules in the system. For example, we know that the persistent data of an object is stored in any virtual system. As a result of the above considerations, Dinopolis objects have to be composed at runtime. This technique named *late composition* is known from component-oriented programming [4] [1]. Hence, the decision was made to develop so-called 'in-house' components (see [4]), which are the object fragments of the Dinopolis object. They are used to build up Dinopolis objects at runtime and it allows to control their life-cycle individually. Regarding the language context our approach focuses on statically-typed (see [6]), main-stream, object-oriented programming languages, namely C++ and Java.

3 The Life-Cycle of Dinopolis Objects

Dinopolis objects have a long-termed life-cycle which is not limited to construction and destruction of objects and it is, for instance, possible to let them "live" as long as their persistent data exists. Moreover, it allows to define at runtime which actions are triggered for each transition in the life-cycle. The basic concept is that Dinopolis objects are built up from scratch starting with an empty template. The empty template provides a method to attach instances of so-called *Definitions.* Definitions are programmed statically (e.g. as a Java class) and they implement one or more *Declarations* (e.g. tagged or marked Java interfaces). Declarations and Definitions are programmed by a module programmer. Among other things, the programmer has to supplement additional information about all implemented Declarations (= the provided interfaces). In the running system, each system module can be asked for an instance of a certain Definition. The module instantiates and initializes the Definition and returns it to the object management module which attaches it to the Dinopolis object. Internally, every Dinopolis object has a table similar to the *virtual method table* created by object-oriented compilers (see [7]). This table, called *explicit virtual table* (EVT) in Dinopolis, is built up at runtime and contains a list of Definitions for each declared method. On the other hand, every object has a list of Declarations which are passed to applications via special proxies (see [8]). We defined the following semantic for attaching an instance of a Definition: When a Definition is attached to the Dinopolis object, first all new Declarations are added to the list of Declarations. Secondly for each declared method the Definition is appended to the list of Definitions. If a method is called on the Dinopolis object, a lookup on the EVT is done and the call is delegated to the most recently appended Definition. This way, the mechanism supports method overriding at runtime.

3.1 Life-Cycle States and Transitions

If a Dinopolis object is newly created the object management module creates an empty skeleton of an object. The only methods available are those of the mechanism for attaching and detaching Definitions. The object management attaches an initial Definition which contains methods for controlling the life-cycle. They are hidden from applications since the life-cycle has to be controlled by the system. Then the Dinopolis object is added to the object management module's internal tables. The first method of the life-cycle is the *creator*. Among other things it determines the correct *static type*. Object creation may include that space for the object's persistent data is allocated in an external system (e.g. a new file is created in a file-system). At the end, the creator calls the *constructor* method on the Dinopolis object. According to the current system configuration, the constructor requests Definitions from the different system modules. Each system module instantiates and initializes its Definitions and returns them to the constructor which, for its part, attaches them to the object. Please note that the object management module may decide at runtime which concrete implementation of the constructor is actually used. When an object is in memory the object management has to handle structural operations. Examples are, moving an object to another location or that the system configuration is changed. This may require that Definitions are attached or detached to/from the object. When the object has been returned to the requestor the object management does distributed reference counting (see [4]) and may decide whether a structural operation can be carried out. Attaching and detaching of Definitions is restricted by several rules. First, there exist implicit rules, which, for example, avoid ambiguities if a method with the same signature is declared in different Declarations. Secondly, the object management has to regard so-called explicit dependencies. Explicit dependencies may exist among different Definitions since Definition programmers can access other Definitions in the same Dinopolis object: Definitions may issue special *scoped* and *supered* calls on other Definitions. Therefore, the programmer has to provide meta-information about these *required* Definitions. With this information, the system may check whether a required Definition is actually attached to the object at runtime. The most powerful functionality provides a method simulating the *this* member variable in C++/Java. It allows programmers to access the entire Dinopolis object. If this method is used the programmer has to provide meta-information about all *required Declarations* which have to be available in a certain Dinopolis object. If no more references to an object exist the object management module may decide to remove it from memory. Therefore, the object management calls the *destructor* on the object. Again it is possible to define at runtime which actions are triggered before an object is removed from memory. At the end of the life-cycle of an object the *deletor* is called. Again, it is possible to define at runtime how object deletion is actually carried out by simply attaching and detaching Definitions.

4 Summary

One of the base requirements of Dinopolis is that is has to be as flexible as possible and that the system can be reconfigured at runtime. Component software technology is applied to support late composition since class-based object-oriented programming is not flexible enough [1]. Object composition, together with a disciplined delegation mechanism, is a possibility to achieve such a high degree of flexibility. The resulting system becomes much more complex as systems using subclassing since it has to ensure integrity and consistency of their objects at runtime. Verifications and checks which are normally done statically by a compiler have to be handled at runtime by the underlying component architecture. For a system using object composition and delegation, a well-defined specification of the life-cycle of objects is essential, since otherwise it is impossible to control such dynamic systems. Moreover, a life-cycle can be exploited to model highly dynamic scenarios which can be found in today's distributed systems. The work presented in this paper is a solution suited for the needs of Dinopolis but we think that the approach of modeling well-known static concepts like method overriding, polymorphism or the destructor is easier to understand for programmers familiar with class-based object-orientation.

References

1. Wolfgang Weck: Inheritance Using Contracts Object Composition, Proceedings ECOOP Workshops, (1997).
2. Klaus Schmaranz: Dinopolis – A Massively Distributable Componentware System, Habilitation Thesis, June (2002)
3. Klaus Schmaranz: On Second Generation Distributed Component Systems, J.UCS Vol. 8, No. 1, 97–116 (2002).
4. Clemens Szyperski: Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 1997.
5. Klaus Schmaranz: DOLSA – A Robust Algorithm for Massively Distributed, Dynamic Object-Lookup Services, submitted to J.UCS.
6. Bertrand Meyer: Object-Oriented Software Construction - Second Edition. Prentice Hall, New York, 1997.
7. Timothy Budd: An introduction to object-oriented programming. Second Edition. Addision Wesley Longman Inc., 1998.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley (1995)

Fighting Class Name Clashes in Java Component Systems*

Petr Hnětynka and Petr Tůma

Distributed System Research Group
Department of Software Engineering, Charles University
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
`hnetynka@nenya.ms.mff.cuni.cz`
`petr.tuma@mff.cuni.cz`

Abstract. This paper deals with class and interface name clashes in Java component systems that occur because of evolutionary changes during the lifecycle of a component application. We show that the standard facilities of the Java type system do not provide a satisfactory way to deal with the name clashes, and present a solution based on administering the names of classes and interfaces with a version identifier using a byte code manipulation tool. We provide a proof of concept implementation.

Keywords: components, Java, versioning, classes, interfaces, name clashes

1 Introduction

Today, the component platforms and the components themselves are often developed in Java [6], where the components are typically sets of Java classes and interfaces. As elements of reuse and integration, the components follow the typical lifecycle of applications [8] and are therefore subject to evolutionary changes. These changes can reach both the internals and the interfaces of a component, and thus, in Java, the corresponding Java classes and interfaces.

In some cases, a change requires different versions of a class or an interface to coexist in one application. This is difficult to do in Java, where two classes or interfaces with the same name cannot coexist unless loaded by unrelated classloaders. In this paper, we describe how to deal with such a situation.

* This work was partially supported by the Grant Agency of the Czech Republic in grants number 102/03/0672 and 201/03/0911.

2 Java Component Systems

We consider contemporary Java component systems such as [2,4,11,13], where the offered services and the context dependencies are represented by a set of provided and required interfaces. Given these interfaces, a component is represented by a set of classes that implement the provided interfaces and call the required interfaces. Besides being implemented directly as a set of Java classes, a component can also be composed from other components tied together through their interfaces.

2.1 Sources of Name Clashes

By a name clash, we understand a situation where a class or an interface cannot be loaded by the Java virtual machine because its name is the same as a name of another class or interface that has already been loaded. We identify three sources of name clashes common to Java component systems. [1]

The first source of name clashes is a situation where the classes that make up a component are replaced by a new version while the component application that uses the component is running. This can lead to a name clash between the old and the new version of the classes.

The second source of name clashes is a situation where the interfaces that make up the component are replaced by a new version but the component application that uses the component does not change and an adapter is used to bridge the mismatch. This can lead to a name clash inside the adapter, which needs to access both the old and the new version of the interfaces.

The third source of name clashes is a situation where a single Java virtual machine is used to run several component applications to avoid redundant loading of shared code. This can lead to a name clash when the component applications use different versions of the same component.

Note that the three sources of name clashes are situations that give rise to a range of other problems besides the name clashes. These are outside the scope of this work, even though some of them are addressed by our proof of concept implementation.

2.2 Standard Solutions and Related Work

Java offers several standard facilities that could be used to solve the problem of class name clashes. These include using unique names, using elaborated classloader hierarchies, and using reflection to connect incompatible but structurally equivalent interfaces. Unfortunately, none of these approaches solves all the considered sources of name clashes and some may lead to performance penalties or inconvenience the programmer, as discussed in [7].

Other solutions to the problem of class name clashes can be found in related work. One system that supports hosting multiple applications within a single JVM

[1] For sake of brevity, we omit the discussion of the likelihood that the situations described here as sources of name clashes will occur in practice. For readers that will not find the practicality of the situations evident, good arguments can be found in [1,8,9,12].

[12] bypasses name clashes by limiting types passed between the applications. Another system that supports dynamic updates through reconfiguration [2] cannot handle updates that involve different versions of the same class or interface. A large class of systems that focus on dynamic adaptation of Java programs [1, 9, 14] solves the problem of class name clashes by using a modified version of JVM. Last, some existing Java component systems [4] simply do not support dynamic updates or coexistence of several versions of a class or an interface with the same name.

3 Removing Name Clashes in Byte Code

Our solution to the problem of class name clashes follows the attractive approach of using unique names for each version of a class or an interface that makes up a component. Unfortunately, this approach inconveniences the programmer, who should be able to use the same names for different versions of the classes and interfaces. A solution to this obstacle rests with byte code manipulation of the executable code of the component application. We use the byte code manipulation to augment the names of classes and interfaces that make up a component by adding a unique version identifier, and to rename the references to these classes and interfaces to use the augmented names. That way, the programmer can use the same names for different versions of the classes and interfaces in a convenient way, without running into name clashes.

3.1 Proof of Concept

As a proof of concept, we have implemented the approach to avoid name clashes through byte code manipulation in SOFA [5, 13]. SOFA is a project of our research group that provides a platform for component applications that supports a construction of hierarchic components connected by potentially complex connectors. The components are fully versioned [10] and can be updated while the component application that uses them is running.

In SOFA, the components are stored in a *template repository* together with the necessary metadata that describe them. When running, the components reside inside a *deployment dock*, which provides the necessary deployment and execution facilities. A single deployment dock can run several component applications. Implemented in Java, it is therefore a platform where all the sources of name clashes that were outlined in Sect. 2.1 can occur.

When a component application is being launched in SOFA, the classes and interfaces are loaded by the deployment dock using a single classloader. To obtain the byte code of the classes and interfaces, the classloader contacts the template repository, which acts as a class server for the deployment dock. The names of the classes and interfaces in the byte code are then augmented using the ASM tool [3] and the augmented version is sent to the deployment dock. The only exception to this mechanism is the case of components that serve as adapters and therefore use several versions of the same interface. In this case, the programmer has to use different names for different versions of the interface simply to be able to write the adapter. The template repository therefore requires an additional translation table that maps the names used by the programmer to the augmented names.

The sole feature of SOFA important for the proof of the concept is the availability of the versioning information, which is used to augment the names of classes and interfaces. In all other aspects, our solution is independent of the SOFA environment. More details and examples can be found in [5, 7].

4 Conclusion

In this paper, we have pointed out the problem of name clashes that occur in Java component systems because of evolutionary changes during the lifecycle of a component application, and outlined the sources of these name clashes. We state that the standard facilities of the Java type system do not provide for a satisfactory solution and may lead to performance penalties and inconvenience the programmer.

We have proposed a solution to the problem of name clashes based on administering the names of classes and interfaces with a version identifier using a byte code manipulation tool. Through a proof of concept implementation, we have also demonstrated that our solution integrates smoothly with a Java component system. Our solution differs from and is superior to the solutions used in contemporary Java component systems.

References

1. Andersson, J.: A Deployment System for Pervasive Computing, ICSM'00, 2000
2. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Composition Framework, http://www.objectweb.org
3. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems, http://www.objectweb.org
4. DeMichiel, L.G., Yalcinalp, L.U., Krishnan, S.: EJB Specification 2.0
5. Distributed Systems Research Group: SOFA, http://sofa.debian-sf.objectweb.org
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification
7. Hnětynka, P., Tůma, P.: Managing Class Names in Java Component Systems with Dynamic Update, TR 2003/2, Dept. of SW Engineering, Charles University, Prague, 2003
8. Lehman, M. M., Ramil, J. F.: Software Evolution in the Age of Component Based Software Engineering, IEE Proceedings Software vol. 147 no. 6, 2000
9. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic Java classes, ECOOP'00, 2000
10. Mencl, V., Hnětynka, P.: Managing Evolution of Component Specifications using a Federation of Repositories, TR 2001/2, Dept. of SW Engineering, Charles University, Prague, 2001
11. Object Management Group: CORBA Components 3.0, OMG, 2002
12. Paal, S., Kammüller, R., Freisleben, B.: Customizable Deployment, Composition, and Hosting of Distributed Java Applications, DOA'02, 2002
13. Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, ICCDS'98, 1998
14. Redmond, B., Cahill, V.: Supporting Unanticipated Dynamic Adaptation of Application Behaviour, ECOOP'02, 2002.

Real-Time Systems Development Using Hierarchical State Machines

Angelo Furfaro, Libero Nigro, and Francesco Pupo

Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS), Italy
a.furfaro@deis.unical.it, {l.nigro,f.pupo}@unical.it

Abstract. This paper proposes a modular design of distributed real-time systems which is based on the Hierarchical Communicating Real-Time State Machines (HCRSM) modelling language, and the Violin toolset. HCRSM combines Statecharts constructs with CSP-like timed communications. Violin provides a visual environment supporting in a seamless way all the life-cycle development phases of an HCRSM system. Temporal validation rests on assertion checking during system simulation. Code generation is based on Java and a customizable runtime support.

1 Introduction

The aim of the work described in this paper is to provide a methodology for the development of distributed real-time systems [21] which is centered on Communicating Real-Time State Machines (CRSM) [19]. CRSM is a formal specification language based on timed finite state machines which interact to one another through synchronous CSP-like channels [11]. CRSM allows the expression of timing constraints on channel communications and internal commands. A CRSM specification is executable for prototyping and temporal testing [16]. Temporal validation rests on assertions [12] on the recorded time stamped event histories of channel communications. Assertion checks are executed during simulation and serve as a temporal testing phase and not as a verification method. Example graphical environments for the editing, prototyping and code generation of CRSM systems have been implemented [16,4]. However, these tools are based on flat or conventional finite state machines which have well-known problems of scalability to large systems.

The contribution of this work is twofold:

– defining HCRSM -Hierarchical CRSM-, a formalism which extends the basic features of CRSM with Statecharts constructs [10,17,18,15]
– implementing in Java the Violin toolset which visually supports the proposed HCRSM based methodology. Violin relies on a runtime representation and associated control engine which favor timing predictability. The runtime representation of a system is used both for prototyping/simulation and code generation in Java.

2 System Architecture

A system specification is organized in two main abstraction levels. The top level, which is referred to as the *architectural* level, models the system as a net of machines connected to one another by means of unidirectional typed channels. The lower level, referred to as the *behavioral* level, models the behavior of each machine.

An HCRSM system model is *closed*, i.e. machines are introduced to model both the computer system and its controlled environment. Each machine owns a set of input/output ports by means of which it communicates with the other machines. Each port it is associated with a tuple of data types (of the target programming language) that constrains the nature of the data that can be received/transmitted through it. A channel links together a pair of *matched* ports owned by two different machines, i.e. an input port and an output port associated with the same tuple of data types.

Ports do not exist in the original CRSM [19]. Their introduction in HCRSM improves modularity as each machine acts as an independent unit of development. Machines share no memory and execute concurrently except when they need to communicate in which case they synchronize their behavior for a message exchange. Synchronous communications [11] through channels are the only way two machines can interact.

3 Behavioral Specification

The architectural level of an HCRSM model is concerned with the decomposition of a system in a set of cooperating machines.

The behavior of each machine, i.e. the way it re-acts to the stimuli coming from its input ports and which can generate signals through its output ports, is specified by a finite state-machine and a set of local data.

Each state transition is labeled by a *guarded-command* and a timing constraint: $G \to C[\tau]$. The guard G is a side-effect free boolean expression that may involve constants and variables of the machine local environment. A necessary condition for a transition to be enabled is that the corresponding guard evaluates to true upon entering the state from which it originates. An omitted guard is assumed to be always true.

The command C can be either an *internal command* or an *IO command.* An internal command models a computation performed by the machine (e.g. expressing an internal processing or a physical activity performed on an interface object like a sensor or an actuator). An IO command specifies either an input or an output operation on a port. An input command has the form `p(v)?`, where `p` is the identifier of an input port of a machine, say `M1`, and `v` is a tuple of variables of the local environment whose types match those of the port specification. An output command has the form `q(expr)!`, where `q` is the identifier of an output port of a machine, say `M2`, and `expr` is an expression that evaluates to a tuple of values whose types match those of the port specification. If `p` and `q` are two matched ports connected by channel `ch`, a communication between `M1` and `M2` through these ports may happen: if they are both ready (*rendezvous*), the couple of IO commands is executed causing a simultaneous state change in both machines and the assignment `v:=expr`.

The timing constraint is a timing interval $[t_{min}, t_{max}], t_{min} \leq t_{max}$ whose interpretation depends on the type of transition. If the lower and upper bounds of this interval are the same, i.e. $t_{min} = t_{max} = t$ it is abbreviated as $[t]$. The time values are relative to the instant when the state, from which the transition originates, is entered.

In the case of an IO command, the timing constraint expresses the possible rendezvous times. Would the time interval be unspecified, it is assumed to be $[0, \infty]$ which means that the machine is ready for communication at any time the partner is. A rendezvous can only occur at any time in the intersection of the intervals associated to the two matching IO commands. An impossible rendezvous can deadlock a machine in the case the transition specifying the IO command is the only one leaving the current state.

The time constraint associated with an internal command models its possible duration. When it is omitted it is assumed to be [0] meaning that the execution of the command takes a negligible amount of time. The upper bound of the interval must be a finite value.

A global time notion is provided in a system which is accessible through the function `rt()`. Each machine owns a special hidden input port called `timer` which can be used to model timeouts. A command like `timer(x)?`$[\tau]$ triggers a timeout to happen after τ time units and its occurrence time to be (possibly) stored in the (optional) local variable `x`.

The bounds of a timing constraint may be specified using expressions involving variables of the local environment and the `rt()` function. Each expression should evaluate to a non negative number, otherwise it is assumed to be 0. Neither of the expressions may produce any side effects.

4 Hierarchical States

The original CRSM [19] formalism uses flat state machines for behavior specification and thus they can not manage the state explosion phenomenon, typical of large systems, which limits the scalability of the approach. Statecharts [8] deal with this problem by introducing a hierarchical model of state machines which allows a higher level of modularity in the behavioral specifications. Statecharts are widely used in the development of reactive systems [10,17,18,5].

Each state of a hierarchical state machine can recursively be decomposed into a set of substates. A state that is not decomposed is said to be a *leaf*, or *basic*, state. The only state that has no parent is called the *root* or *top* state.

Statecharts define two types of state decomposition: *or*-decomposition and *and*-decomposition. In the former case a state is broken down into a set of substates which are in an “exclusive-or” relation, i.e. if at a given time the state machine is in a compound state it is also in exactly one of its substates. In the latter case substates are related by logical "and", i.e. if the state machine is in the parent state it is also in all of its substates each of which acts as independent concurrent component. For simplicity, HCRSM admits only the *or*-decomposition and thus the machine is the unit of concurrency.

At a given point in time, a machine's behavior can find itself simultaneously in a set of states that constitutes a path leading from one of the leaf states to the top state. Such a set of states is called *configuration* [9]. A configuration is uniquely characterized by the only leaf state which it contains.

Each decomposed state S specifies which of its substates must be considered its *initial state*. This substate is indicated by means of a curve originating from a small solid circle and ending on its border (*default transition* of S).

Both source and destination of a transition can be states at any level of the hierarchy. Whereas a transition always originates from the border of a state, it can reach its destination state either on its border or ending on a particular element called *history connector* or H-connector. Such a connector is depicted as a small circle containing an H (*shallow history*) or an H^* (*deep history*), and it is always inside the boundary of a compound state.

Firing a transition leads the machine behavior to move from one configuration to another. When a configuration is left each belonging non *leaf* state keeps memory of its direct substate that is also part of the configuration. This substate is referred to as the *history* of the compound state. The first time a state is entered, its history coincides with its initial state.

Let S be the destination state of a transition tr. The configuration which is assumed as consequence of firing tr depends on the way tr reaches S.

- If S is a leaf state the new configuration is the only one that contains S.
- If S is a compound state and tr ends on its border, the next configuration corresponds to the destination state being the initial state of S.
- If S is a compound state and tr ends on a shallow history connector, the next configuration corresponds to the destination state being the state that is history of S.
- Finally, if S is a compound state and tr ends on a deep history connector, the configuration depends on the nature of the state D which is currently history of S. If D is a leaf state, the configuration will be the only one that contains D, otherwise the configuration corresponds to the case tr would end on a deep history connector of D.

As an example, consider the state machine in Fig. 1 where the current configuration is $C_s = \{Top, B, B1, B12\}$, and the history of A consists of state$A2$ and within it $A22$. In case of firing of the transition going from B to the H^*-connector of A, the configuration changes from C_s to $C_d = \{Top, A, A2, A22\}$.

To each compound state is associated an *initialization action* which is a sequence of assignment statements that can change the values of the variables of the local environment. The initialization action is executed each time the corresponding compound state is entered by taking its default transition, i.e. when it is entered by a transition ending on its border.

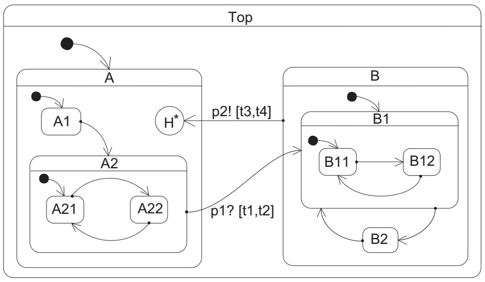


Fig. 1. A hierarchical state machine

In order to reduce visual cluttering, the internal decomposition of a compound state can be omitted and displayed in a separate diagram. In this case, a transition that crosses the boundary of a compound state, is displayed, in different diagrams, using particular connectors called *state ports.*

5 Operational Semantics

At any point in time, the status of each machine is determined by its current configuration and by the values of the variables of its local environment. The global status of a system consists of the statuses of all its component machines.

The evolution of a system can be described as a sequence of steps each one corresponding to a change in the global status and (possibly) of the global time. A step can involve a configuration change in a machine, caused by an internal command or a timer firing, or the simultaneous change of the configurations of two communicating machines, due to the execution of a pair of IO commands on two matching ports.

Let *tr* be a transition. The notation $tr.\underline{tw}$ indicates the static time window associated to *tr*, and $tr.\underline{tw}.lb$ and $tr.\underline{tw}.ub$ denote respectively the lower and the upper bound of the time window. A time window *tw* is said to be absolute if it has been evaluated at current (or enabling) time: $tw.lb = \underline{tw}.lb + rt()$, $tw.ub = \underline{tw}.ub + rt()$. For simplicity, in the following it will be implicitly assumed that a transition time window is handled after being made absolute.

Transition *tr* is *enabled* if its guard evaluates to true. Transition *tr* is *ready*, i.e. it *can* fire, if it is enabled and $rt() \in tr.tw$.

An enabled internal command is automatically ready. Since the associated timing constraint models only the possible durations of the command, its *readiness* corresponds to an implicit interval $[0, \infty]$.

Enabling and readiness of a transition are evaluated each time the state from which the transition originates is entered. Transition is disabled when this state is left off.

The initial configuration of every machine is set up by firing a virtual transition without source that ends on the border of the machine top state. At system start-up every machine reaches "instantaneously" its initial configuration. As a result, a set of states are entered and a set of transitions enabled.

The abstract HCRSM executor is naturally event-driven. An event captures one transition (internal command or timer) or a couple of simultaneous transitions (a rendezvous) to fire, plus a time interval. The time interval coincides with the transition time window in the case of an internal command or a timer, or to the intersection of the two involved time intervals for a rendezvous. The executor repeats a basic control loop. At each iteration (next step), the most imminent event, if there are any, is selected. The set of candidate events is the so called *proposed set* which contains all the events associated to the enabled transitions emanating from the states of the current (e.g., initial) configurations. A rendezvous event requires the enabling of the two IO commands involved and a non empty intersection of their time windows. The event selection process is guided by the Earliest Time First strategy (ETF). All the events having the same minimal lower bound are first identified then one event is chosen non deterministically. Let *next* be the chosen event. The system clock is first adjusted to $max(rt(), next.tw.lb)$. The executor goes on by removing from the proposed set all the events (*revoked set*) which are associated to enabled transitions exiting from the states which will be abandoned after *next* execution. Then, the action associated to *next* is performed, i.e. one or two state transitions are carried out. Finally, the proposed set is changed by adding any new events to it corresponding to the entered states, and the control loop is repeated.

The control loop terminates when the proposed set empties, i.e. the system is found to be deadlocked.

In order to characterize the set of states that are respectively entered and exited at a configuration change, two semantic models, called *flat* and *modular*, are considered (see Fig. 2). Flat semantics takes the point of view that all the states which are part of the leaving configuration are exited and then all the states of the new configuration are entered. The approach is called flat because it views each different configuration as a state of a corresponding plain state machine. The approach implies that the enabling of transitions originating from

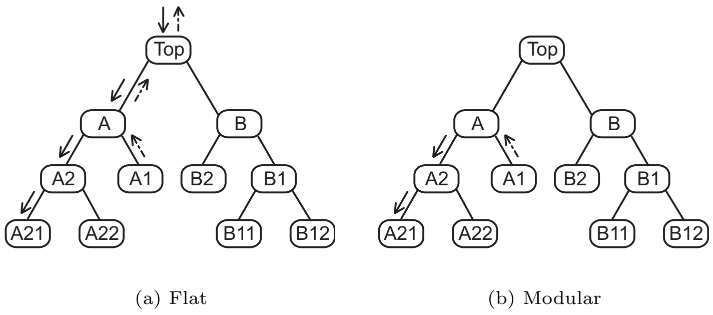


Fig. 2. Enter and exit sequences under flat and modular semantics

the states of the reached configuration, along with their timing constraints, are always re-evaluated with respect to the current time horizon.

Under modular semantics, a more conservative approach is taken: only the states that are in the levels crossed by the transition are involved in configuration switching. This is the same approach adopted by Statecharts semantics [9] which introduces the concept of a transition *scope*. In HCRSM, the scope of a transition tr is defined as the lowest common ancestor in the hierarchy of states that properly contains both source and target state of tr. When tr fires, all the states of the leaving configuration that are substates of its scope are exited, and all the states of the entering configuration which are substates of the scope are entered.

More formally, let C_s and C_d be the sets of states that are respectively part of the abandoned and the reached configuration, and S_{exit} and S_{enter} the sets of states respectively exited and entered. Flat semantics defines $S_{exit} = C_s$ and $S_{enter} = C_d$.

Definitions under modular semantics are slightly more complex. Let A_S be the set of states that contains state S and all of its proper ancestors. It follows that:

$$\begin{aligned} S_{exit} &= C_s \setminus K \\ S_{enter} &= C_d \setminus K \end{aligned} \quad \text{with} \quad K = A_{src} \cap A_{dst}$$

where src and dst are respectively the state from/to which the firing transition originates/terminates.

As a consequence of modular semantics, some events are not removed from the proposed set because the enablings of the relevant transitions are not changed. All of this allows a modular interpretation of the timing constraints of transitions outgoing from a compound state S because they are not affected by transitions whose scope does not contain S.

Figure 2 illustrates the states which are entered and exited, under flat (a) and modular (b) semantics, when transition from $A1$ to $A2$ in Fig. 1, fires. In this case, source and destination configurations are respectively $C_s = \{Top, A, A1\}$ and $C_d = \{Top, A, A2, A21\}$, the scope of the transition is state A and under modular semantics $S_{exit} = \{A1\}$ and $S_{enter} = \{A2, A21\}$. It is worth noting that the transition from B and ending on the H^*-connector of A is unaffected in the modular approach by the configuration switch and thus its timing constrain remains relative to the time the B state was last entered.

6 Temporal Testing through Assertions

Given the model of an HCRSM system the problem of automatically verifying whether it satisfies a set of not trivial properties is usually not tractable. The high-level language used for expressing actions typically implies an infinite set of execution states. The situation gets even worse when properties involve timing constraints.

The approach adopted in this work for functional and temporal validation is based on testing: each property is checked at *interesting* points during system simulation. Obviously, the approach does not ensure that a given property is always verified for each possible system execution, but it gives a certain degree of confidence about system behavior under typical execution scenarios.

Properties of a system can be stated by reasoning about machine communications, i.e. messages exchanges over channels. Therefore, in order to explore meaningful system behavior, channel communications must be traced. For each channel a copy of the transmitted data and the occurrence time of the event are recorded on the so called *Time stamped Event History* (TEH) [20] associated with the channel:

$$e_0(d_0)@t_0 \; e_1(d_1)@t_1 \; e_2(d_2)@t_2 \; \ldots$$

where e_i is the i^{th} message in the history carrying data d_i and t_i is its occurrence time. The history of channel communications can be generated by executing the specification and examined through *assertions* based on the RTL (Real Time Logic) formalism. An assertion is triggered by an event occurrence and can exploit the `time(channel,index)` and `value(channel,index)` functions which respectively return the occurrence time (or -1 if it does not exist) and the data value of the message occurrence specified by `index`. A positive index selects an event from the beginning of the history and negative index applies from the latest event in the `TEH`. In particular, the value -1 indicates the last event. An assertion is logically specified in the following form:

```
when <event> [ + <delay> ] {
  assert( <RTL_expression> )
}
```

where `<RTL_expression>` is an RTL expression, `<event>` is the trigger event and `<delay>` is an optional positive value that specifies how many time units after the occurrence of the trigger the RTL expression should be evaluated. The `assert()` command can simply write to a text file the kind, the occurrence time and the boolean result of the checked assertion. Assertions can capture safety conditions, i.e. the set of system conditions which should never occur. They can also serve for deadline checking, i.e. testing that an event representing the response to an external stimulus is provided, under different load conditions within a certain deadline. An HCRSM model is translated into executable code and exploits a runtime system which is customized for simulation or real time execution. It provides support for assertion checking.

7 The Violin Toolset

Violin is an integrated graphical development environment that leverages the power of HCRSM. It is totally written in Java and may thus be used on any Java-enabled platform. The main features of Violin are: graphical editing of HCRSM diagrams, checking of topological properties, assertion specifications for system validation, automatic code generation/compilation for system prototyping and final implementation. Figure 3 shows the graphical user interface of Violin.

The features of Violin can be accessed by menu selection with the most common ones also available through the buttons of the *command bar* located under the menu bar. The developer can associate a project with a single machine

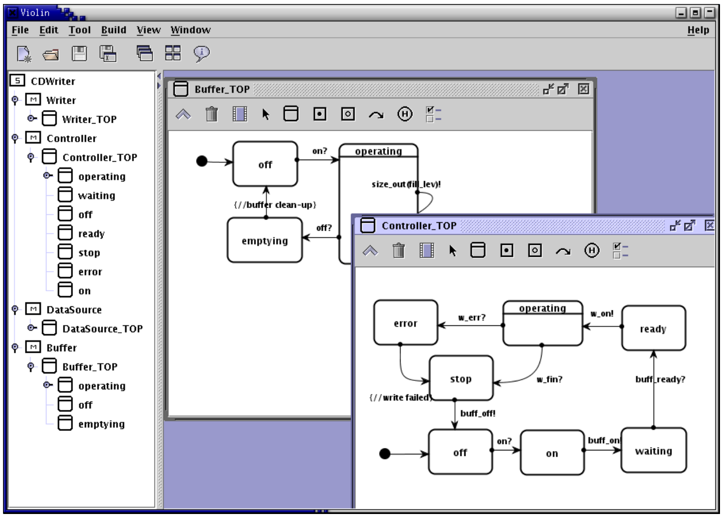


Fig. 3. Violin GUI

or a full system. The main window of Violin is split in two parts: on the left side there is a tree-view that shows the hierarchical decomposition of current project, on the right side there is a *desktop pane* which can host multiple internal windows each showing the details of a corresponding node of the tree-view. The graphical editing capabilities rely on jDraw, a Java reusable framework for the development of domain specific graphical editors. jDraw was inspired by Unidraw [22] and was purposely developed to support the Violin editing features. Violin uses a library for higraphs manipulation built on top of jDraw to represent the hierarchical model of an HCRSM system.

Patterns [7] are extensively used in the design of Violin, e.g, support for automatic code generation, topological checking and XML externalization are based on the Visitor pattern. All of this brings a great level of reusability and extensibility, e.g. code generation can easily be changed/extended to support different runtime systems, languages and compilers. Currently, code generation is targeted to Java. HCRSM diagrams can be exported to various graphical formats (JPEG, PNG, SVG) to be easily included in other documents.

7.1 Control Engine

A key factor of Violin is the adoption of a runtime representation of an HCRSM system which, under flat or modular semantics, is used both for prototyping and for final implementation.

Machines are mapped on thread-less reactive objects with atomic actions. The runtime representation is interpreted by a control engine which is event-driven, time-sensitive and based on non-preemptive scheduling. The control engine runs in one thread of the hosting Operating System. Its behavior reproduces the basic control loop of the abstract executor described in Sect. 5. Event processing (i.e. transition firing) extends the control thread. The control loop is resumed at event termination. The approach minimizes dependencies from the underlying OS and contributes to timing predictability [14]. In addition, the runtime system purposely avoids Java dynamic memory allocation and garbage collection. Objects (e.g. states, events, channel data, etc.) are pre-allocated at system start-up and reused dynamically.

The control strategy of the engine can be specialized to behave as simulation or real time execution. The control engine relies on an *event list* where events concerning transition firings (of a rendezvous, internal command or timer) are scheduled/descheduled respectively when they are part of the *proposed* or *revoked set* (see Sect. 5). The event list is kept ranked according to ETF.

For real execution, the engine uses a "real" time notion built on top of a hardware clock system. Would the $rt()$ value be less than the lower bound of the most imminent event with time window $[t_1, t_2]$, the control engine simply waits for the real time to advance towards t_1.

For simulation, a pseudo-time notion is used which is advanced by a discrete amount at each control step. For instance, dispatching an internal command with duration $[d_1, d_2]$ causes the simulation time to be advanced by a quantity d, $d_1 \leq d \leq d_2$. For worst case analysis, $d = d_2$. Firing a timer with absolute expiration time t causes the simulation time to be set to t. Dispatching a rendezvous with time interval $[t_1, t_2]$ can imply first advancing the clock to t_1 in the case $rt() < t_1$. A rendezvous actually consists of two messages which are delivered to the two partner machines. To avoid zero time increment in cycles of state transitions, a minimal time (the *dwell-time* parameter of the simulator) is supposed to be spent by any machine in any state.

Dispatching a rendezvous first causes all assertions which registered on the triggering channel to be executed. Assertions are uniformly represented as special-case machines equipped with suitable behavior. States can be introduced for achieving delayed execution of assertion code, using the timer facility.

Actually, the Violin simulation tool is distributed in character and makes it possible to partition the machines of a system into a set of logical processes (LPs) so as to control the degree of concurrency. Each LP is assigned a group of one or more machines and is served by a local event list and control loop. The various LPs interact with one another and are synchronized by a global control structure which conservatively ensures that the system clock is always incremented by the minimum quantity. Transition firings affecting distinct LPs are concurrent, whereas transitions within a same LP are fired sequentially. As an example, the multi-LP simulation can be exploited for testing a distributed implementation of an HCRSM system, with a deterministic communication network (e.g. CAN [3]) which provides guarantees about the transmission delay of network messages. In this example, each LP is associated with a physical processor and relies on an I/O network board (IONB) machine which receives local communications and maps them onto network messages and vice versa. For the purposes of prototyping/simulation the network itself can be abstracted by a machine which has connections only with IONB machines and introduces the proper transmission delay to each network message. In the concrete system implementation an IONB module directly interfaces with a network physical board through a suitable low-level driver.

8 Conclusions and Outlook

This paper proposes a methodology for building systems with timing constraints which is based on Hierarchical Communicating Real-Time State Machines. The design of HCRSM was driven by timing predictability. Only a subset of Statecharts constructs were included, which facilitate implementation and customization of the runtime infrastructure. A toolset –Violin– was developed in Java for modelling, prototyping, checking properties through assertions and simulation, and implementing real-time systems. In its current version, Violin supports Java code generation of a modelled system.

On-going work is geared at:

- providing more formal semantics to HCRSM;
- supporting model checking [2] of HCRSM systems, by mapping a model on to Timed Automata [1] in the context of the Uppaal tool [13]. Currently, a translation from CRSM to Uppaal was defined, in the presence of some restrictions, e.g. using integer variables with an associated limited range of values. The goal is unfolding an HCRSM model into a basic CRSM one and then exploiting the established mapping for verification purposes;
- improving the graphical editor of Violin, e.g. with parameterized super states [8] in order to enhance behavioral reusability in state machines;
- extending Violin with different runtime systems and target languages;
- experimenting with the practical use of the methodology in complex time-critical systems and in the modelling, evaluation and implementation of multimedia sessions with quality of service control over the Internet [6].

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] A. Burns. How to verify a safe real-time system: The application of model checking and timed automata to the production cell case study. *Real-Time Systems*, 24(2):135–151, March 2003.

[3] CAN. A serial bus - not just for vehicles. In *Proc. of 1st Int. CAN Conference (ICC'94)*, 1994.

[4] G. Fortino and L. Nigro. A toolset in Java2 for modeling, prototyping and implementing communicating real-time state machines. *Microprocessors and Microsystems*, 23(3):573–586, 2000.

[5] G. Fortino, L. Nigro, F. Pupo, and D. Spezzano. Super actors for real time. In *Proc. of 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, pages 142–149, January 2001.

[6] A. Furfaro, L. Nigro, and F. Pupo. Multimedia synchronization based on aspect oriented programming. *Microprocessors and Microsystems*, to appear.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, July 1987.

[9] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[10] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts.* McGraw-Hill, 1998.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[12] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.

[13] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1/2):134–152, 1997.

[14] L. Nigro and F. Pupo. Schedulability analysis of real time actor systems using coloured petri nets. In G. Agha, F. D. Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets – Advances in Petri Nets*, LNCS, pages 493–513. Springer, 2001.

[15] Object Management Group. OMG Unified Modeling Language specification, version 1.4. http://www.rational.com/uml.

[16] S.C.V. Raju and A.C. Shaw. A prototyping environment for specifying and checking communicating real-time state machines. *Software–Practice and Experience*, 24(2):175–195, February 1994.

[17] B. Selic, G. Gullerkson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* Wiley Professional Computing. John Wiley & Sons, Inc., 1994.

[18] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. http://www.rational.com/media/uml/resources/documentation/umlrt.pdf, 1998.

[19] A.C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, 1992.

[20] A.C. Shaw. Time-stamped event histories: a real-time programming object. In *Proc. of 22nd IFIP/IFAC Workshop on Real Time Programming (WRTP'97)*, pages 97–100, Lyon, September 1997.

[21] A.C. Shaw. *Real-Time Systems and Software.* Wiley, 2001.

[22] J.M. Vlissides and M.A. Linton. Unidraw: a framework for building domain-specific graphical editors. *ACM Transactions on Information Systems (TOIS)*, 8(3):237–268, 1990.

Classboxes: A Minimal Module Model Supporting Local Rebinding

Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts

Software Composition Group, University of Bern
Bern, Switzerland
{bergel,ducasse,wuyts}@iam.unibe.ch

Abstract. Classical module systems support well the modular development of applications but do not offer the ability to add or replace a method in a class that is not defined in that module. On the other hand, languages that support method addition and replacement do not provide a modular view of applications, and their changes have a global impact. The result is a gap between module systems for object-oriented languages on one hand, and the very desirable feature of method addition and replacement on the other hand. To solve these problems we present *classboxes*, a module system for object-oriented languages that provides method addition and replacement. Moreover, the changes made by a classbox are only visible to that classbox (or classboxes that import it), a feature we call *local rebinding*. To validate the model, we have implemented it in the Squeak Smalltalk environment, and performed experiments modularising code.

Keywords: language design, method lookup, modules, smalltalk, class extension, selector namespace

1 Modules in the Presence of Extensibility

The term *module* is overloaded. We follow the definitions of Modular Smalltalk [16] and Szyperski [12].

Modules are program units that manage the visibility and accessibility of names. A module defines a set of constant bindings between names and objects [16]. A module is a capsule containing (definitions of) items. The module draws a strong boundary between items defined inside it and items defined outside other modules [12].

A *class extension* is a method that is defined in another source packaging entity (for example, a Java package or an Envy application [9]) than the class it is defined for. There exist two kinds of class extension: a *method addition adds* a new method, while a *method replacement replaces* an existing method.

Classical module systems, like those of Modula-2[17], Modula-3 [1], Oberon-2 [8], Ada [13], or MzScheme's [4] do not support class extensions. Numerous object-oriented programming languages, such as Java, C++, and Eiffel [7] lack this facility. However, it is widely used in those languages that support it, such as Smalltalk[16] and GBeta [3]. In "Capsules and Types in Fresco" A. Wills reports that in the goody library[1] goodies-lib@cs.man.ac.uk 73% of the files modify existing classes, and 44% define no new classes at all [14]. Even if these figures should be tempered due to the fact that goodies are not industrial applications, these numbers reflect that class extensions are not an anecdotical mechanism. There is some ongoing research that explores the introduction of class extensions to Java (for example OpenClasses [2], Keris [18] or MixJuice [5]), which is another indication that this is quite an important concept.

Languages supporting class extensions such as Smalltalk or Flavors do not offer the notion of modules. In these languages the changes are globally visible and impact the whole system. Even in module systems that support class extensions (Modular Smalltalk [16]), changes are visible to everyone after they have been applied.

To summarise, module systems exist for languages that do not support class extensions on the one hand, and languages exist that support class extensions but not modules on the other hand. The Classbox model provides modules that fully support class extensions, and these extensions are only visible to the classbox that defined them. Outside the classbox the system runs unchanged. This is accomplished by redefining the method lookup mechanism to take classboxes into account, so that the desired method is executed.

For validation we implemented this system in Squeak, an open-source Smalltalk environment, and implemented some small applications. Section 3 describes one of these examples, an application to check dead links on a web page. Classboxes are used to extend an existing system with a visitor and to replace existing system code.

The rest of the paper is structured as follows. Section 2 presents an overview of the Classbox model. In Sect. 3 we illustrate the model by showing the implementation of an application to check for dead links on web pages. Section 6 concludes the paper.

2 Overview of the Classbox Model

This section describes the semantics of the Classbox model. The next section illustrates the semantics and usage on a concrete case-study highlighting its unique features.

Classbox Contents. A classbox consists of *imports* and *definitions*:

- An import is either a *class import* (stating explicitly from which classbox the class is imported, called the *parentbox*) or a *classbox import* (*i.e.,* that imports every class from the imported classbox).
- A definition can be a class definition or a method definition. A method definition declares the class that a method belongs to, the name of the method, and the implementation of the method.

[1] A goody is a small application provided without warranty or support.

Static Completeness. Methods can only be defined on classes that are known within the classbox (*e.g.,* defined or imported). Furthermore, the implementation of a method can only refer to classes known in the classbox.

Extension. Extending a class C with one method m has the following semantics: if the class C has a method with the same signature as m, m replaces that method, otherwise m is added to C.

Flattened Class. A flattened class describes what methods a class in a certain classbox contains, taking imports into account:

– The flattened definition of a class C *defined* in a classbox cb1 consists of C and all the method definitions for C in cb1.
– The flattened definition of a class C *imported* in a classbox cb1 is the *flattened* definition of C in its parentbox extended by the method definitions for C in cb1.

Note that this implies that for the method lookup, importing takes precedence over inheritance (first the import chain is used, and then the inheritance chain). This is explained in Sect. 4.

Flattened Classbox. A flattened classbox consists of the flattened definitions of all the classes (defined or imported) of that classbox.

Class Name Uniqueness. When defining or importing a class C in a classbox cb1, the name of C has to be uniquely defined in flattened C. This guarantees that class import cycles are not possible.

Method Addition. Method m is a method *addition* for class C if m is a method defined for C, and the *flattened* definition of C in its parentbox does not define a method with the same signature as m.

Method Replacement. Method m is a method *replacement* for class C if m is a method defined for C, and the *flattened* definition of C in its parentbox contains a method with the same signature as m. Following the definition of flattening, the method replacement takes precedence in the flattened version of C.

These rules ensure the following property, which we call *local rebinding.* Suppose a classbox cb1 defines a class C with two methods, m calling n, and classbox cb2 imports C from cb1 and replaces n. We say that cb2 locally rebinds n in cb1 to represent the fact that calling m in the context of cb2 invokes the method n as defined in cb2 while calling m in the context of cb1 invokes the method n as it is defined in cb1. The runtime semantics are illustrated in the next section and explained in Sect. 4.

3 The Running Example

To illustrate the key properties of the Classbox model we develop an application that allows one to check dead links on a web page. We use *Squeak*[6], an open source Smalltalk, to implement the Classbox model. The user specifies the web

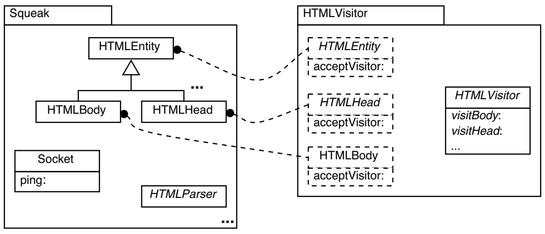


Fig. 1. The Squeak Classbox and the HTMLVisitor Classbox that extends Squeak with an HTML Visitor

page to be checked and the application returns the list of URLs that cannot be reached within a given time-out.

Out of the box, the Squeak environment comes with a sophisticated development environment and a rich class library. All this code is contained in a single "image" within a single global namespace and consists of about 1800 classes. Squeak contains a HTML parser, a hierarchy of HTML nodes that are built by the HTMLParser and several network protocols.

To write our application we use the existing HTML parser to create a HTML tree of the web page for which we want to check the dead links. Then we walk this tree, checking whether each link can be reached. While these checks can be hard-coded as methods in the HTML parse tree itself, it is a common practice to write a visitor for the HTML parse tree which can be reused by other applications. Then we only need to write a specialised visitor that checks for dead links. Therefore, the implementation of our application consists of the definitions of two classboxes: one extending Squeak with a visitor for the HTML parse tree and one customising that generic visitor with one that checks for dead links. The resulting system is shown in Fig. 2, and is explained in the next sections.

3.1 Class Import and Class Extensions

As shown in Fig. 1, extending the HTML parse tree with a visitor consists in adding a new HTMLVisitor class, and adding one method to each existing HTML parse tree node to call the visitor. Since the visitor methods and the visitor class itself logically belong together, we group them in the same classbox called HTMLVisitor. Figure 1 shows a part of the Squeak classbox (that contains the whole unmodularised library of around 1800 classes of the Squeak environment) and the HTML visitor classbox. This classbox imports every HTML parse tree node class (only three are shown in the picture) and extends each of these classes with a single method to visit them (called acceptVisitor:). It also contains the HTMLVisitor class, that implements the abstract visitor class.

Illustrated Model Properties: Method Additions. The example shows that classboxes can be used not only to define whole classes (like the HTMLVisitor class); they can also define methods on classes that are imported (all the acceptVisitor: methods), *i.e.*, classboxes support method additions [16] [2].

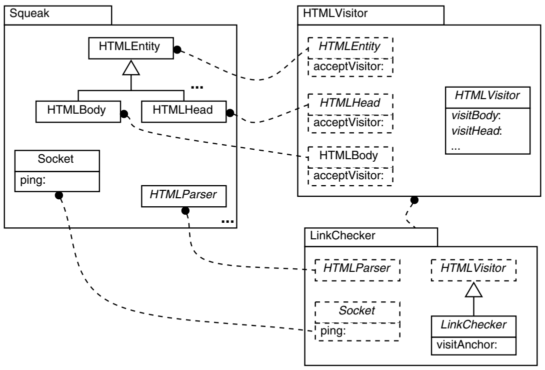


Fig. 2. The LinkChecker classbox, that defines a HTMLVisitor subclass that checks for dead links and that replaces the method ping: in the class Socket to throw exceptions instead of opening dialogue boxes

Without this feature it is very difficult to factor out the visitor in a separate classbox from the tree it operates on. In languages that do not support method additions, the solution would be to create visitable subclasses for every HTML node and add the visiting methods there. However this has two undesirable effects: first of all, the existing code that used the classes has to be changed to use the new subclasses, and second, other applications that need to make similar extensions would have to independently add the same subclasses. Class extensions do not exhibit either problem.

Illustrated Model Properties: Local Rebinding. The addition of the method acceptVisitor: to the HTML tree classes is only visible for code executed in the context of the HTMLVisitor classbox. Code running in the Squeak classbox cannot see this method addition. The next sections elaborate on this point.

3.2 Classbox Import and Method Replacement

With the HTMLVisitor classbox defined it becomes easy to implement the Link Checker application. It basically boils down to adding a subclass of HTMLVisitor that overrides the visitAnchor: method to implement the checking of dead links. Checking a link means opening a connection to the specified URL within a certain amount of time.

In practice it turns out that the class that actually builds the connection (the class Socket) does not throw exceptions when links are not reachable. Instead it directly opens a dialogue box explaining the error that was encountered!

This makes it suddenly quite hard to implement the Link Checker. The solution would be to change the method that opens a dialogue box and let it throw exceptions instead. However, this also means that all the applications that use this method and rely on dialogue boxes to be opened have to be changed as well.

While this may be a worthwhile endeavour that would result in a cleaner Squeak system, it is too much work when just writing a Link Checker. The solution is to change this method in such a way that it will throw exceptions only in the places where it is needed. All other places should still use the unchanged method. This is what is done by the LinkChecker classbox. How this works is explained in detail in Sect. 4.

Figure 2 shows the classbox LinkChecker. It imports the classbox HTMLVisitor, *i.e.*, all classes defined in HTMLVisitor are imported and it imports class Socket from the Squeak classbox. The classbox LinkChecker defines a class LinkChecker, a subclass of HTMLVisitor, and a method visitAnchor: that implements the actual checking of the links contained in a HTML document. It also defines a method ping: on Socket that replaces the existing implementation that opens dialogue boxes with an implementation that throws exceptions.

Illustrated Model Properties: Local Rebinding. This example shows that a classbox allows one to replace methods for existing classes. Moreover, these changes are again *local* to LinkChecker: it is only in the LinkChecker classbox that exceptions are raised when network locations are not reachable. The rest of the system is unaffected by this change, and still gets dialogue boxes when time-outs occur.

3.3 Local Rebinding and Flattening

This section elaborates how local rebinding in the presence of the flattening property allows a classbox to change the behaviour of methods in the system in such a way that these changes are *local* to the classbox.

To illustrate this we execute some expressions in the example described in previous sections. We execute an expression that creates an HTML parse tree for a certain url in two different contexts: first in the Squeak classbox then in the LinkChecker classbox.

```
HtmlParser parse: ('http://www.iam.unibe.ch/~scg/' asUrl contents)
```

In the Squeak classbox, the result of this expression is a parse tree consisting of instances of HTMLEntity that cannot be visited. This is exactly the intended behaviour, as the expression is performed in the context of the Squeak classbox, and that classbox does not know anything about visitors for its parse tree.

In the LinkChecker classbox the same expression the result is a HTML parse tree that can be visited. Again, this is exactly what is intended since we imported the HTML parse tree nodes from HTMLVisitor (indicating that we want those classes to be used).

4 Runtime Semantics of the Model

Depending on the classbox an expression is executed in, objects can understand different messages or have methods with different behaviour. For this to work,

```
 1  lookup: selector class: cls
 2              startBox: startbox currentBox: currentbox classboxPath: path
 3
 4      | parentBox theSuper togoBox newPath |
 5      self
 6          lookup: selector
 7          ofClass: cls
 8          inClassbox: currentbox
 9          ifPresentDo: [:method | ^ method].
10      parentBox := currentbox providerOf: cls name.
11      ^ parentBox
12          ifNotNil: [path addLast: parentBox.
13                     self
14                        lookup: selector
15                        class: cls
16                        startBox: startbox
17                        currentBox: parentBox
18                        classboxPath: path]
19          ifNil: [theSuper := cls superclass.
20                  theSuper ifNil: [^ cls method: selector notFoundIn: cls].
21                  togoBox := path  detect: [:box | box scopeContains: theSuper].
22                  newPath := togoBox = startbox
23                                 ifTrue: [OrderedCollection with: startbox]
24                                 ifFalse: [path].
25                  self
26                      lookup: selector
27                      class: theSuper
28                      startBox: startbox
29                      currentBox: togoBox
30                      classboxPath: newPath]
```

Fig. 3. The lookup algorithm that provides local rebinding

a classbox-aware lookup mechanism for methods and a change in the structure of method dictionaries are needed. We focus on Smalltalk method dictionaries here, but the same approach holds for other object-oriented languages.

Normally, method dictionaries are used to lookup a key consisting of the signature of the method (in Smalltalk this is only the name, as there are no static types), and return a value corresponding to the method body. To support classboxes we encode the classbox where the method is defined in the method signature (*i.e.*, the key). For example the method dictionary for HTMLEntity has entries prefixed with "#Squeak." for the methods defined in Squeak, and entries with "#HTMLVisitor." for the method additions defined in that classbox. The method dictionary for class Socket now has two entries for the ping: method: one for the Squeak classbox and one for the LinkChecker classbox. Class LinkChecker has only a single entry for the visitAnchor method.

Encoding the classbox with the method signature makes it possible to let different implementations for a method live alongside each other. However, to take advantage of this, the method lookup mechanism has to be changed as well. Figure 3 describes the lookup algorithm we implemented that ensures the local rebinding property.

The algorithm first checks whether the class in the current classbox implements the selector we are looking for (lines 5 to 9). If it is found, the lookup is successful and we return the found method (line 9). If it is not found, we recurse.

Table 1. Benchmarks results from Squeak comparing the regular method lookup mechanism with the classbox-aware lookup mechanism (units are in milliseconds)

Benchmark	**Regular lookup**	**Classbox lookup**	**Overhead**
direct call	5439	6824	25%
looked up call	5453	6940	27%
opening and closing a web browser	332	548	65%
opening and closing a mailreader	536	760	41%
call through 3 classboxes	-	10554	-
call through 6 classboxes	-	10654	-

The algorithm favours imports over inheritance, meaning that first the import chain is traversed (in lines 12 to 18) before considering the inheritance chain (in lines 19 to 30). This last part is the difficult part of the algorithm, since we need to find the classbox where the superclass is defined that is closest to the classbox we started the lookup from. Therefore the algorithm remembers the path while traversing the import chain (line 12), and uses this when determining the classbox for the superclass (line 21).

4.1 Runtime Performance

As can be expected, introducing the classbox aware method lookup mechanism introduces some runtime overhead, especially since our current implementation is currently not optimised. Table 1 shows the results for some benchmarks that we performed to compare the regular method lookup performance vs. the classbox-aware lookup performance:

1. sending a message defined in the class of the instance (10 millions times), and sending a message defined in a super class hierarchy (10 millions times).
2. measuring launching and closing of two applications implemented in Squeak (a web browser and an e-mail client) within the same classbox (average over 10 times).
3. performing a method call through a chain formed by classboxes extending a class.

The table shows that the penalty for the new lookup scheme by itself is roughly 25 percent, where the real-world applications run about 60 percent slower. We think that this difference is due to the fact that we did not adapt the method cache in the virtual machine.

Note that our current implementation is straightforward and does not incorporate any optimisations yet. For example we are thinking of changing the structure of the method cache in order to take classboxes into account.

5 Related Work

No existing mainstream language supports class extensions, modules, and local rebinding. Classical module systems, like those of Modula-2[17], Modula-3 [1], Oberon-2 [8], Ada [13] or Java, do not support class extensions.

Keris introduces extensible modules which are composed hierarchically and linked implicitly. Keris does not support class extension [18]. MzScheme's units are modules system with external connection facilities [4], and act as components that can be instantiated and linked together. They do not support class extensions. Classboxes on the other hand, are source code management abstractions.

OpenClasses [2] supports a modular definition of class extensions but only supports method addition and not method replacement. MixJuice [5] offers modules based on a form of inheritance which combines module members and class extensions but not local rebinding.

Modular Smalltalk only supports method additions that are globally visible [16]. In the Subsystems proposal [15], modules (subsystems) support selector namespaces, as in SmallScript [10]. A selector namespace contains method definitions (of possibly different classes). Selector namespaces are nested and this structure is used for the method lookup.A local selector takes precedence over the same selector defined in a surrounding namespace. With selector namespaces, class extensions can be defined as layers where methods defined in a nested namespace may redefine methods defined in their enclosing namespaces. Selector namespaces, however, do not support local rebinding.

Us, a subject-oriented programming extension of Self [11], provides for object extensions and method invocations in the context of *perspectives*, but Us does not provide modules.

6 Conclusion

This paper introduces the Classbox Model, a module model for object-oriented systems that supports local rebinding. Hence it provides method additions and replacements that are only visible in the module that defines them. Classboxes enhance both existing object-oriented languages that support method addition and replacement, and those that provide module systems. For the former it localises method additions and replacements. It extends the latter with a mechanism that supports unanticipated evolution. To apply local rebinding to an object-oriented language efficiently, the method lookup mechanism has to be changed, and a slightly different method dictionary has to be introduced.

We have implemented the model in the Squeak Smalltalk environment, and performed experiments using classboxes. In the paper we describe an example of how classboxes allow one to extend an existing parse tree with a visitor (making use of class extensions), and replacing a badly implemented method in a system class without affecting the whole system (using method replacement). As far as we know, no other module system is able to achieve this separation.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

References

1. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, Aug. 1992.
2. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000*, 130–145, 2000.
3. E. Ernst. Propagating class and method combination. In *Proceedings ECOOP '99*, volume 1628 of *LNCS*, 67–91, June 1999. Springer-Verlag.
4. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the PLDI '98 Conference on Programming Language Design and Implementation*, 236–248, 1998.
5. Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, June 2002. Springer Verlag.
6. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, 318–326, Nov. 1997.
7. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
8. H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.
9. J. Pelrine, A. Knight, and A. Cho. *Mastering ENVY/Developer*. Cambridge University Press, 2001.
10. D. Simmons. Smallscript, 2002. `http://www.smallscript.com`.
11. R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.
12. C. A. Szyperski. Import is not inheritance – why we need both: Modules and classes. In *Proceedings ECOOP '92*, volume 615 of *LNCS*, 19–32, June 1992. Springer-Verlag.
13. S. T. Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, volume 28, 127–143, Oct. 1993.
14. A. Wills. Capsules and types in fresco. In *Proceedings ECOOP '91*, volume 512 of *LNCS*, 59–76, July 15–19 1991. Springer-Verlag.
15. A. Wirfs-Brock. Subsystems – proposal. OOPSLA 1996 Extending Smalltalk Workshop, Oct. 1996.
16. A. Wirfs-Brock and B. Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, 123–134, Nov. 1988.
17. N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 1983.
18. M. Zenger. Evolving software with extensible modules. In *ECOOP 2002 International Workshop on Unanticipated Software Evolution*, June 2002.

Zonnon for .NET – A Language and Compiler Experiment

Jürg Gutknecht and Eugene Zueff

Computer Systems Institute, ETH Zürich, Switzerland
{gutknecht,zueff}@inf.ethz.ch

Abstract. In this article we introduce and present a new programming language called Zonnon and its implementation for .NET. Zonnon is an evolution of Oberon. It distinguishes itself by a wide spectrum of applicability and by a highly comprehensive object model. Special highlights are an advanced notion of active object and a unified concept of abstraction called definition. We also briefly discuss both the mapping of Zonnon's object model to .NET and the use of a new compiler integration technology called CCI.

1 Introduction

This is a report on work in progress. The project emerged from our participation with Oberon in the academics part of Project 7, an initiative launched in 1999 by Microsoft Research with the goal of exploring language interoperability on the .NET platform. Our motivation to continue was driven by two different forces: a) the desire to further explore the potential of .NET and b) the desire to experiment with evolutionary language concepts.

We first present the design principles of our new programming language codenamed *Zonnon*. Then, we discuss some aspects of mapping Zonnon's enhanced object model to .NET and, finally, sketch a generic implementation of the Zonnon for .NET compiler.

At the time of writing this text it is too early for drawing final conclusions from this experiment. First steps with using the new language to implement selected samples of programs showed that Zonnon is very close to Oberon along the "programming-in-the small" axis but very far from Oberon along the "programming-in-the-large" axis.

2 The Zonnon Language

Zonnon is an evolution along the Pascal, Modula, Oberon language line [1]. Its design has been guided by the following principal objectives:

– **Versatility in Terms of Programming Paradigms.** The language should seamlessly cover a variety of programming levels and styles. If used for the development of software of any of the following categories, it should neither impose a methodological "corset" nor require unnatural trains of thought:

  - algorithms and data structures
  - modular and embedded systems
  - object oriented systems
  - component systems
  - actor and agent systems
- **Uniform Concept of Abstraction.** The language should provide one and only one generic notion of abstraction. It should refrain from conceptually distinguishing similar kinds of abstractions or abstractions whose differences are caused by artefacts alone. In particular, the language should unify the two concepts of (abstract) base class and interface.
- **Effective Compositional Framework.** The language should support and encourage to a largest possible degree the reuse of existing software components for the construction of new and more complex systems.

On the background of these principles, we are now well prepared to present the design of the Zonnon language and the rationale of its constructs.

There are four different kinds of selfcontained program units in Zonnon: *Objects*, *modules*, *definitions* and *implementations*. The first two are runtime units, the third is a unit of abstraction and the fourth is a unit of reuse. Here is a brief characterization of each of Zonnon's program units:

- **Object.** Template of a selfcontained runtime unit. May include intrinsic behavior expressed in terms of one or more activities run by separate threads. Can be instantiated dynamically under program control in arbitrary multiplicity.
- **Module.** Both container of related declarations and object whose life cycle is controlled exclusively by the system. Typically a singleton.
- **Definition.** Abstract description of a *facet* of an object. Typically containing declarations of types, constants, variables and method signatures (in this article we use the term *method* synonymously with *procedure*)
- **Implementation.** Reusable partial or total implementation of a definition. In the following sections, we take an exemplary approach to explaining the four kinds of program unit and their use in more detail and breadth, where we consciously choose caricaturelike simplified illustrations. For a detailed definition of the Zonnon language, we refer to the language report [2]

2.1 Objects and Modules

Compared to Oberon, the notion of *object type* has been upgraded in Zonnon and made explicit by the use of the keyword OBJECT (instead of RECORD). In addition to fields, Zonnon objects may feature *methods* and *activities*. While methods specify the services offered to clients by instances of this type, activities describe their intrinsic behavior. Each activity is run by a separate thread that is created and started automatically as a side effect of each instance creation. Note that the concept of activities in Zonnon is similar to the concept of active bodies in *Active Oberon*, an earlier evolution of Oberon presented in [3] and [4].

However, activities are superior in two respects: first, they may declare (thread-) local data and second, they may be composed to a more complex, concurrent behavior of individual objects.

A good example of active objects is provided by the *pipeline* design pattern. Each element of a pipeline is both a link in a chain of processing stations and an activity transforming arriving objects into a new state of production.

Another example are creatures called *Zoilath* in some phantasy world. Their behavior is this: While the temperature is below a certain minimum, Zoilaths simply hibernate, otherwise they either take a random walk or, if they are hungry, hunt for prey.

```
OBJECT Zoilath (x, y, t: INTEGER);
  VAR X, Y, temp, hunger, kill: INTEGER;
  PROCEDURE SetTemp (dt: INTEGER);
    BEGIN { LOCKED } temp := temp + dt
  END SetTemp;
  ACTIVITY;
    LOOP
      AWAIT temp >= minTemp;
      WHILE hunger > minHunger DO
        HuntStep(5, kill);
        hunger := hunger - kill;
        WHILE (kill > 0) & (hunger > 0) DO
          HuntStep(7, kill);
          hunger := hunger - kill
        END;
        RandStep(2)
      END;
      RandStep(4); hunger := hunger + 1
    END
  END
BEGIN X := x; Y := y; temp := t; hunger := 0
END Zoilath;
```

Note that the *activity* part of the object declaration coherently tells the full life story of these creatures. Further note that their behavior still depends crucially on the “weather maker” in the environment calling the `SetTemp` method (in mutual exclusion, of course). In particular, every instance entering the loop is blocked by the `AWAIT` statement until the temperature is reported to have risen above the limit. Further note that the initialization statements in the object body guarantee each creature to start its active life in a well defined state, depending on the object parameters passed to the creator.

While Zoilaths may exist in large number in our hypothetical terrarium and new instances can be created dynamically at any time, there is only one environment. However, this does not mean that it is an unstructured monolithic block. Quite the reverse, the environment is typically a conglomerate of separate parts or *modules*, where each module roughly corresponds to a certain service. Clearly, some services may *import* (use) others. For example, assume that modules *Food*, *Map* and *Weather* exist as part of our environment, where *Food* imports *Map* and textitWeather, and *Map* imports *Weather*:

```
MODULE Food;
  IMPORT Map, Weather;
  PROCEDURE { PUBLIC } GetLocation (VAR X, Y: INTEGER);
  BEGIN
    IF Weather.isWet(X, Y) THEN Map.GetLake (X, Y)
      ELSE Map.GetMountain (X, Y)
    END
  END GetLocation;
END Food;

MODULE Map;
  IMPORT Weather;
  VAR ...
  PROCEDURE GetNextLake (VAR X, Y: INTEGER);
  PROCEDURE { PUBLIC } GetLake (VAR X, Y: INTEGER);
  BEGIN GetNextLake(X, Y);
    WHILE Weather.isIcy(X, Y) DO GetNextLake(X, Y) END
  END GetLake;
  PROCEDURE { PUBLIC } GetMountain (VAR X, Y: INTEGER);
END Map;

MODULE Weather;
  PROCEDURE { PUBLIC } isWet (X, Y: INTEGER): BOOLEAN;
  PROCEDURE { PUBLIC } isIcy (X, Y: INTEGER): BOOLEAN;
  ...
END Weather;
```

The `PUBLIC` modifier denotes visibility of the item to importing clients. If absent (as in the case of *GetNextLake* in module *Map*), the item is hidden to clients. Figure 1 gives a graph representation of the modular system just discussed.

Modules in Zonnon are components of the runtime environment. Basically, a Zonnon module at runtime is a “static” object that is loaded on demand and managed by the system exclusively. Historically, the concept of software module was introduced long ago by Hoare (in the context of concurrency and under the name of *monitor*) and Parnas, and it was soon supported by “modular” languages, for example by Mesa, Cedar, Modula-2, Ada and Oberon. Unfortunatley, it later lost its state of a first-class citizen and was relegated to an object design pattern called the “singleton pattern”. We strongly believe that, in combination with the *import* relation, the module construct is a powerful system building tool that deserves explicit support by a modern programming language. We should add that a module/import graph designates the structure of a runtime system

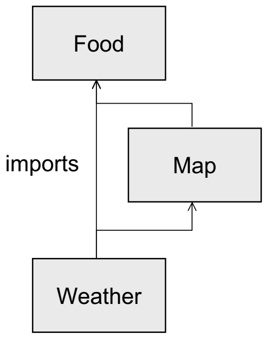


Fig. 1. Example of a modular system

and should neither be confused with an object oriented inheritance diagram nor with a dynamic data structure fabricated at runtime.

2.2 Definitions and Implementations

A *definition* is an abstract view on an object from a certain perspective or, in other words, an abstract presentation of one of its *facets*. For example, a jukebox has two facets. We can look at it alternatively as a record store or as a player. The corresponding definitions are:

```
DEFINITION Store;
  PROCEDURE Clear;
  PROCEDURE Add (s: Song);
END Store.

DEFINITION Player;
  VAR current: Song;
  PROCEDURE Play (s: Song);
  PROCEDURE Stop;
END Player.
```

Assume now that we have the following *default implementation* for the `Store` definition

```
IMPLEMENTATION Store;
  VAR rep: Song;
  PROC Clear;
    BEGIN rep := NIL
  END Clear;
  PROC Add (s: Song);
    BEGIN s.next := rep; rep := s
  END Add;
BEGIN Clear
END Store.
```

Then, we can obviously aggregate the `Store` implementation with the jukebox object:

```
OBJECT JukeBox IMPLEMENTS Player, Store;
  IMPORT Store; (* aggregate *)
  PROCEDURE Play (s: Song); IMPLEMENTS Player.Play;
  PROCEDURE Stop IMPLEMENTS Player.Stop; ...
  PROCEDURE { PUBLIC } AcceptCoin (value: INTEGER);
END JukeBox.
```

It is worth noting that an arbitrary number of implementations can be aggregated with an object in this way. We also emphasize that implementations need not be complete but may leave methods unimplemented. The rule is that, in this case, it is mandatory for the implementing object to provide an implementation, while in all other cases (re)implementation is optional.

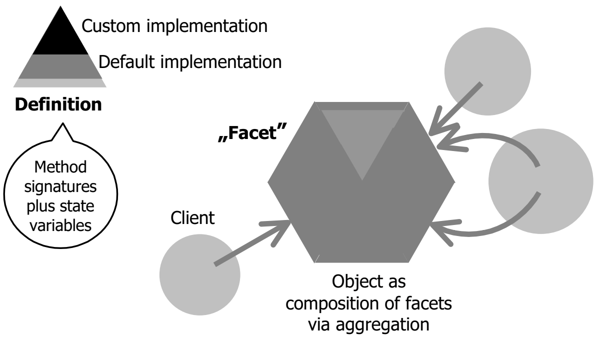

Custom implementation
Default implementation
Definition
Method signatures plus state variables
„Facet"
Client
Object as composition of facets via aggregation

Fig. 2. Definition: unifying abstraction concept

Figure 2 depicts our object model. Zonnon objects present themselves to external observers as multifacetted entities, with an *intrinsic facet* (the set of public members) and a set of *extrinsic facets* that are specified in form of definitions. Internally, Zonnon objects consist of a kernel, possibly including active behavior specification, and an implementation for each definition. Every implementation splits into a default part and an object specific part, where the default part is simply aggregated with the object.

Depending on their perspective, clients access Zonnon objects either directly via some public method (corresponding to the intrinsic facet perspective) or indirectly via definition. For example, if

```
VAR j: Jukebox; s: Song;
```

is a declaration of a Jukebox instance and a song instance, then `j.AcceptCoin(5)` is a direct access to the Jukebox intrinsic facet, while `Player(j).Stop` and `Store(j).Add(s)` are indirect accesses to `j` via definitions.

Declarations of object instance variables are either *specific* or *generic*. The above declaration of a juke box is specific. A corresponding generic variant would simply declare the juke box as an object with a postulated implementation of the definitions `Player` and `Store`:

```
VAR j: OBJECT { Player, Store } ;
```

Unlike in the above case of a specific declaration, it would obviously not make sense to address the Jukebox's intrinsic facet.

Let us conclude this section with some additional comments including fine points and future perspectives.

- As a fine point we first note that there need not be a one to one correspondence between definitions and implementations. It is well possible for an object to expose a definition without aggregating any default implementation. In turn, an object may aggregate an implementation without exposing the corresponding definition (if any exists at all).
- Our concept of definition and implementation subsumes and unifies two concepts of abstraction that are separated artificially in languages as Java and C#: Base class and interface. While the base class abstraction is generously supported by these languages in the form of implicit inheritance, the interface abstraction is not, and its use necessitates the delegation of the work to "helper classes". We consider this asymmetry a highly undesirable artefact from the perspective of software design, because it requires the designer to distinguish one abstraction as the "primary" one. What is the primary abstraction in the case of our juke box, what in the cases of objects that are both container and applet, figure and thread etc. ?
- Our model of aggregation resembles multiple inheritance as used in some objectoriented languages such as C++ and Eiffel. However, there are two significant differences. First, our implementation aggregates are never instantiated autonomously in the system, they correspond to abstract classes. Second, our inheritance is explicit by name rather than implicit, protecting the construction from both name conflicts and obscure or unexpected method selection at runtime.
- The concept of definition is a principle rather than a formal construct. It is a door to extending the language without compromising its core part. Two examples:
  - We may liberally introduce a *refinement* relation between definitions that, in its simplest form, reduces to extension. However, more elaborate forms might support stepwise refinement by allowing entire refinement chains of some component specification.

- In future versions of the language, we may open the syntax and accept, for example XML element specifications as definitions. On this basis it would definitely be worth revisiting XSLT transformation schemes and "code behind" techniques for active server pages.

2.3 Small Scale Programming Constructs

With the design of the Zonnon language, we have intentionally concentrated on the large scale aspects of programming and have largely kept untouched the basic syntax and style of the Modula/ Oberon family.

With one exception, however: we added a *block statement* with optional processing modifiers in curly braces that allows us to syntactically unify similar but semantically different cases. For example, method bodies and critical sections of activities within an object scope both use the `LOCKED` modifier:

```
BEGIN { LOCKED } ... END
```

Other, more future oriented modifiers are `CONCURRENT` and `BARRIER` to signal the option of statement-parallel processing within the containing statement block. And finally, the block construct allows us to include *exception catching*, a first and important concession to interoperability on the .NET platform, a topic that we shall continue to discuss in the next section:

```
BEGIN { ... } ... ON exception DO ... END
```

3 Mapping Zonnon to .NET

While we consciously kept the design of Zonnon independent of the possible target platforms of its implementation, the aim of implementing the new language on the .NET platform can hardly be hidden. We already mentioned *exception catching* as a concession to interoperability. Another one is *namespaces*. However, unlike C#, Zonnon does not know an extra namespace construct. It simply allows qualified names to occur everywhere in the global program scope, where all but the last part are used to identify the name space.

A much more intricate topic is the smooth integration of Zonnon's object model into .NET or, equivalently, its mapping to the .NET *Common Type System (CTS)* [5]. Focusing on the nonstandard part of the object system, the constructs that essentially must be mapped are *definition*, *implementation*, *object activity*, and *module*. In this overview, we restrict our discussion to some remarks on the mapping of definitions and implementations.

Different options exist. If state variables in definitions are mapped to *properties* or *virtual fields* (given the latter exist), the complete state space can theoretically be synthesized in the implementing object, albeit with some access efficiency penalty. In contrast, the solution suggested below in terms of C# (.NET's canonical language) is based on an internal helper class that provides the aggregate's state space.

```
DEFINITION D;
  TYPE e = (a, b);
  VAR x: T;
  PROCEDURE f (t: T);
  PROCEDURE g (): T;
END D;

IMPLEMENTATION D;
  VAR y: T;
  PROCEDURE f (t: T);
  BEGIN x := t; y := t
  END f;
END D;
```

is mapped to

```
interface D_i {
  T x { get; set; }
  void f(T t); T g (); }

internal class D_b {
  private T x_b;
  public enum e = (a, b);
  public T x {
    get { return x_b };
    set { x_b = ... } } }

public class D_c: D_b {
  T y;
  void f(T t) {
    x_b = t; y = t; }  }
```

4 The Zonnon for .NET Compiler

4.1 Compiler Overview

The Zonnon compiler has been developed for the .NET platform and itself runs on top of it. It accepts Zonnon compilation units and produces conventional .NET assemblies containing MSIL code and metadata. Further, the compiler is implemented in C# and uses the Common Compiler Infrastructure framework (CCI) that has been designed and developed by Microsoft.

4.2 The Common Compiler Infrastructure

The CCI framework provides generic support for compiler writers targeting at .NET. More concretely, the CCI supports the construction of an intermediate graph representation (henceforth called IR) from source code and a series of transformations, ending with MSIL code. It also supports the full integration of the compiler and of other software development components into Visual Studio for .NET.

The CCI basically consists of two sets of (C#) classes:

– a rich hierarchy of IR node classes representing the typical constructs of the Common Language Infrastructure (CLI), including *class*, *method*, *statement*, *expression* etc.
– a collection of classes representing "visitors" that support important IR transformations such as resolving overloaded identifiers, consistency checking and code generation.

Compiler writers now simply reuse the IR node classes provided by the CCI to internalize the corresponding constructs of their own language and, if necessary, they add new IR classes for unsupported constructs. While in the former case the visitor classes provided by the CCI can be reused for transformation purposes, each new IR class must be accompanied by corresponding visitors, either extensions of some standard CCI visitor or completely new visitors.

The CCI supports five kinds of visitors:

– Looker
– Declarer
– Resolver
– Checker
– Normalizer

All visitors walk through the IR performing various kinds of transformations: The *Looker* visitor (together with its companion *Declarer*) replaces *Identifier* nodes with the members or locals they resolve to. The *Resolver* visitor resolves overloading and deduces expression result types. The *Checker* visitor checks for semantic errors and tries to repair them. Finally, the *Normalizer* visitor prepares the intermediate representation for serializing it to MSIL and metadata.

All visitors are implemented as classes inheriting from CCI's *StandardVisitor* class. Alternatively, the functionality of a visitor can be extended by adding methods for the processing of specific language constructs, or a new visitor can be created. The CCI requires that all visitors used in a compiler (directly or indirectly) inherit from *StandardVisitor* class.

Integration Service is a variety of classes and methods providing integration into the Visual Studio environment. The classes encapsulate specific data structures and functionality that are required for editing, debugging, background compilation etc..

4.3 The Zonnon Compiler Architecture

Conceptually, the organization of our compiler is quite traditional: the *Scanner* transforms the source text into sequence of lexical tokens that are accepted by

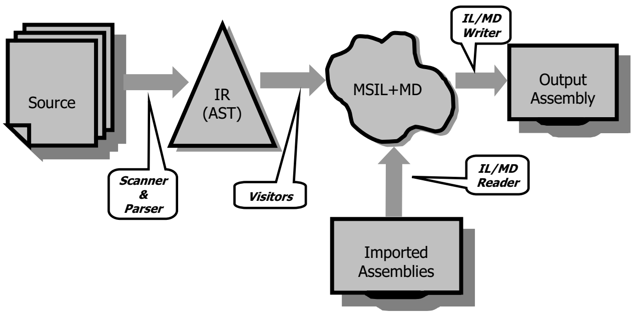


Fig. 3. Zonnon compilation model

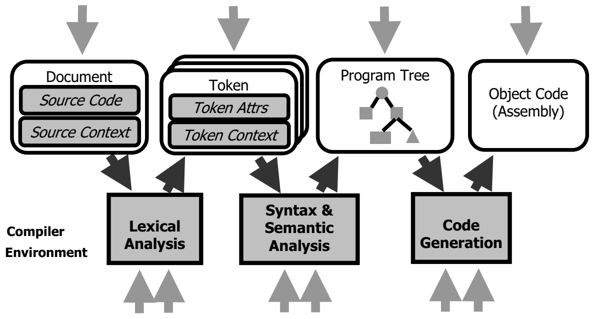


Fig. 4. Zonnon compiler architecture: compiler as a collection of resources

the *Parser*. The Parser is implemented by recursive descent. It performs syntax analysis and builds an abstract syntax tree (AST) for the compilation unit using IR classes. Every AST node is an instance of an IR class. The semantic part of the compiler consists of a series of consecutive transformations of the AST built by the Parser to a .NET assembly. Figure 3 shows the Zonnon compilation model.

From an architectural point of view, the Zonnon compiler differs from most of the "conventional" compilers. The Zonnon compiler is no "black box" but presents itself as an open collection of resources. In particular, data structures such as "token sequence" and the "AST tree" are accessible (via a special interface) from outside of the compiler. The same is true for compiler components. For example, it is possible to invoke the Scanner to extract tokens from some specified part of the source code, and then have the Parser build a subtree for just this part of the source. Figure 4 illustrates the overall Zonnon compiler architecture.

This kind of open architecture is used by the CCI framework to provide a natural integration of the compiler into the Visual Studio environment. The CCI contains prototype classes for Scanner and Parser. The actual implementations of Scanner and Parser are in fact classes inheriting from the prototype classes.

Our compiler extends the IR node set by adding a number of Zonnon specific node types for notions such as *module*, *definition*, *implementation*, *object* and for some other constructs having no direct counterpart in the CCI. The added nodes are processed by the extended *Looker* visitor which is a class inherited from the standard CCI *Looker* class. The result of the extended *Looker's* work is a semantically equivalent AST tree, however restricted to nodes of predefined CCI types. In particular, it is the extended Looker that implements the mappings described in Sect. 3.

Acknowledgments. The authors gratefully acknowledge the opportunity of collaborating with Microsoft in this project. In particular, we thank Herman Venter for providing the CCI framework and letting us be his Guinea pigs. Our sincere thanks also go to Patrik Reali for his valuable advice concerning the mapping of Zonnon definitions to .NET. We also greatly appreciated the valuable comments by the reviewers of an earlier version of this text. And last but definitely not least, we acknowledge Brian Kirk's good naming taste. *Zonnon* could in fact turn out to be more that just a code name.

References

1. Wirth, N. The Programming Language Oberon. In *Software – Practice and Experience 18:7.* pp. 671–690, July 1988.
2. Kirk, B., Lightfoot, D. *The Zonnon Language Report.* 2003.
3. Gutknecht, J. Do the Fish Really Need Remote Control?, A Proposal for SelfActive Objects in Oberon. In *JMLC'97* pp. 207–220
4. Reali, P. *Using Oberon's Active Objects for Language Interoperability and Compilation.* Diss. ETH No. 15022
5. Gutknecht, J. Active Oberon for .NET: An Exercise in Object Model Mapping. In *BABEL'01, Satellite to PLI'01.* Florence, IT, 2001.

Safely Extending Procedure Types to Allow Nested Procedures as Values[*]

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. The concept of nested procedure values, i. e., the possibility of using nested procedures as values of procedure types, is a useful and powerful concept. Nevertheless, it is not allowed in languages such as Modula-2 and Oberon(-2), because it creates a serious security hole when used inappropriately. To prevent such misuse while at the same time retaining the benefits of the concept, alternative language rules as well as a small language extension for Oberon-2 are suggested, which allow nested procedures to be safely used as values of procedure types and especially to pass them as parameters to other procedures.

1 Introduction

Nested procedures, i. e., procedures declared local to another procedure, are a useful concept for structuring and decomposing large procedures into smaller and more comprehensible pieces in a natural way, without needing to introduce artificial global procedures to achieve that aim. Furthermore, the fact that nested procedures can directly access the local variables and parameters of their enclosing procedures helps to keep their parameter lists short, without needing to introduce artificial global variables for that purpose.

Procedure types, i. e., types possessing procedures as values, are another useful concept for program development that allows algorithms (e. g., for sorting) to be decoupled from their basic operations (e. g., comparing objects) and by that means increasing their generality and applicability.

The *combination* of nested procedures and procedure types, i. e., the possibility of using not only global, but also nested procedures as actual parameters of other procedures, would be an even more useful and powerful concept, as the example given in Sect. 2 will demonstrate. Unfortunately, however, languages such as Modula-2 [9] and Oberon(-2) [10, 6] do *not* allow this combination, i. e., they require procedure values (i. e., values of procedure types) to be global procedures. After explaining in Sect. 3 the most important reason for this apparently strange restriction, i. e., the problem of *dangling procedure values*, Sect. 4 suggests alternative language rules which *do* allow nested procedure values (i. e., nested procedures as procedure values) without running into this problem. Since there remains at least one important application of procedure values that is permitted

[*] An extended version of this paper is available as a Technical Report [3].

by the original rule, but not by the new ones, a simple language extension is suggested in Sect. 5 to overcome this limitation, too. The paper closes in Sect. 6 with a discussion of the proposed rules and of related work.

2 An Example of Nested Procedure Values

To give a simple example of nested procedures values, Fig. 1 shows a procedure `Trav` that recursively traverses in infix order a binary tree `t` containing integer values, executing a callback procedure `cb` for every node's value. In many applications of this procedure, it would be natural to use a nested procedure as callback procedure because of its ability to access local variables of its enclosing procedure. For example, Fig. 2 shows a procedure calculating the sum of all values stored in the tree `t` by calling procedure `Trav` with the nested procedure `Add` as callback procedure.

```
TYPE
  Tree = POINTER TO Node;
  Node = RECORD
    val: INTEGER;
    left, right: Tree;
  END;
  CallbackProc = PROCEDURE (x: INTEGER);

PROCEDURE Trav (t: Tree; cb: CallbackProc);
BEGIN
  IF t # NIL THEN
    Trav(t.left, cb);
    cb(t.val);
    Trav(t.right, cb);
  END
END Trav;
```

Fig. 1. Traversing a binary tree

3 Rationale for Disallowing Nested Procedure Values

Unfortunately, nested procedures are *not* allowed as values of procedure types in languages such as Modula-2 and Oberon(-2) causing the above example to be actually *illegal*. When considering the usefulness of the concept, this appears to be a completely unreasonable restriction at first glance. In the tree traversing example, for instance, it would be extremely unnatural to declare `Add` as a global procedure because this would require to declare the variable `sum` globally, too.

However, there is at least one important reason justifying this restriction, even though it is rarely explained in language reports or textbooks: Passing

```
PROCEDURE Sum (t: Tree) : INTEGER;
  VAR sum: INTEGER;

  PROCEDURE Add (x: INTEGER);
  BEGIN sum := sum + x
  END Add;
BEGIN
  sum := 0;
  Trav(t, Add);
  RETURN sum;
END Sum;
```

Fig. 2. Application of procedure `Trav`

nested procedures as parameters to other procedures or assigning them to variables would be similar to passing around pointers to local variables; and just like dereferencing such a pointer after the procedure containing the local variable has exited would be illegal, calling a nested procedure via a procedure variable after the enclosing procedure has exited, would be illegal, because the local variables of the enclosing procedure, which might be accessed by the nested procedure, do no longer exist. In analogy to dangling pointers, such procedure values will be called *dangling procedure values* in the sequel.

4 Alternative Language Rules for Oberon-2

In order to retain the benefits of nested procedure values without creating the danger of dangling procedure values, assignments of procedure values to "more global" variables must be forbidden. This can be achieved by replacing the original rule:

R0: Procedure values must be global procedures.

with the following set of definitions (L1 to L3) and rules (R1 to R3):

L1: As usual, the *lifetime of a variable* is defined as the execution time of its directly enclosing procedure. To simplify terminology, a module is treated as a top-level procedure for that purpose.

Of course, the lifetime of an array element or record field is identical to the lifetime of the enclosing array or record.

The lifetime of an explicitly or implicitly dereferenced pointer is defined as the execution time of the program, because a pointer always refers to dynamically allocated storage whose lifetime extends to the end of the program as long as it is referenced by at least one pointer.

L2: Likewise, the *lifetime of a procedure name* is defined as the execution time of its directly enclosing procedure.

That means in particular, that the lifetime of a procedure name is quite different from the execution time of a particular activation of this procedure. The former actually represents the time where the procedure can be correctly and safely invoked.

L3: A *procedure value* is either (i) a procedure name or (ii) the value of a procedure variable or (iii) the result of calling a procedure whose result type is a procedure type, either directly by its name or indirectly via a procedure variable. In all these cases, the *lifetime of a procedure value* is defined as the lifetime of the procedure name or variable used to obtain the value.

R1: The assignment of a procedure value to a procedure variable (i. e., a variable of a procedure type) is forbidden, if the variable's lifetime exceeds the value's lifetime.
Passing a procedure value as an actual parameter is treated like an assignment of the value to the corresponding formal parameter, i. e., formal parameters are treated like variables.

R2: Returning a procedure value from a procedure is forbidden, if the lifetime of the returning procedure's name (not the execution time of the current procedure activation!) exceeds the value's lifetime. In particular, a procedure must not return a local procedure name or the value of a local procedure variable.
This rule is in accordance with the above definition of the lifetime of a procedure value that is obtained from a procedure call (L3).

R3: The assignment of a procedure value to a `VAR` parameter is treated like returning that value, i. e., the lifetime of the enclosing procedure's name must not exceed the value's lifetime.
Conversely, passing a procedure variable as an actual parameter for a `VAR` parameter is treated like an assignment of the called procedure's result value (if it would be of the correct type) to that variable, i. e., the variable's lifetime must not exceed the lifetime of the called procedure.

Since procedure variables and values might be embedded in records or arrays, the above rules must be applied to these accordingly. Furthermore, it should be noted, that the above rules do not replace, but rather augment the other rules of the language. For example, in addition to rule R2, the general rule that the value returned by a procedure must be assignment compatible with the procedure's result type, has to be obeyed.

Normally, the relative lifetimes of two "objects" (procedure names or variables) are determined by lexical scoping: Objects declared in the same procedure obviously have equal lifetimes, while the lifetime of an object declared in a more global procedure exceeds the lifetime of an object declared in a more local procedure. (The lifetimes of objects declared in unrelated procedures never have to be compared.) As a special additional case, however, the lifetime of an actual parameter value always exceeds the lifetime of the corresponding formal parameter (which is identical to the execution time of the called procedure). Together with rule R1, this observation implies the important corollary that procedure values can be passed as parameters *without any restriction.*

5 An Additional Language Extension

Under these rules, the example given in Sect. 2 is correct, because procedure values are only passed as parameters there and never stored in any other variables.

Unfortunately, however, rule R1 forbids the implementation of modules that encapsulate a global procedure variable by a pair of procedures to get and set the variable, respectively, because the set procedure is not allowed to assign the value of its parameter to the global variable. On the other hand, if clients would only pass global procedures to the set procedure, no dangling procedure values would actually occur. To express such a requirement, it is possible to enhance a procedure variable with a *lifetime guarantee* by extending its type with an optional *lifetime guarantee clause* `OF` *`name`* after the keyword `PROCEDURE` to express that variables of that type must not contain procedure values whose lifetime is shorter than the execution time of procedure *`name`*. In other words, only procedures declared in procedure *`name`* or in more global procedures are allowed as values of such variables. As a special case, it is also possible to use the keyword `MODULE` instead of the module's name to express that the value of a variable must be a global procedure.

To generalize the rules stated in Sect. 4 to variables with lifetime guarantees, the term "lifetime" has to be replaced with "lifetime guarantee," whose exact definition is given in [3].

6 Related Work and Conclusion

After describing the dilemma that nested procedure values are on the one hand very useful, but on the other hand forbidden in Oberon-2 and related languages for a serious reason (dangling procedure values), alternative language rules as well as a small language extension have been proposed to retain their benefits without the danger of running into trouble.

It is interesting to see in this context, that the problem solved in this paper does not even appear in many other programming languages. For example, standard C [4] and C++ [7] do not allow nested procedures at all, so the notion of nested procedure *values* is simply not applicable. (C++ *function objects* are a completely different matter; they can be used to some extent to simulate nested procedure values.) Breuel [2], however, describes a corresponding extension for C++ that is implemented in the GNU C (but interestingly not in the GNU C++) compiler. Following the typical style of C and C++, however, the problem of dangling procedure values (just as the problem of dangling pointers) is not addressed at the language level, but left to the programmer, i. e., he is responsible for using the concept correctly. Here, the approach presented in this paper could be applied in the same way as for Oberon-2 to make the language safer.

Likewise, Eiffel [5] does not provide nested procedures, while other object-oriented languages such as Java [1] do not provide procedure (resp. method) values at all, so *nested* procedure values are not appropriate either. (Eiffel *agents* are again a completely different matter, comparable to C++ function objects.)

Functional languages such as Lisp, ML, or Haskell [8] fully support nested functions including *closures*, which are just another name for nested procedure values. But since pure functional languages lack the notion of variables to which closures might be assigned, the problem of dangling closures can only appear when a function returns a locally defined function, which is forbidden in the present paper by rule R2, but typically allowed in functional languages. In such a case, the runtime system takes care to retain the environment of a function (i. e., the non-local values it accesses) as long as necessary, even after the enclosing function has exited.

When comparing the alternative language rules suggested in this paper with the original rule:

R0: Procedure values must be global procedures.

the former appear to be much more complicated than the latter. For most practical applications, however, the simple rule of thumb:

RT: Procedure values must not be assigned to more global variables.

is sufficient, while the rules given in Sect. 4 are just a more precise and complete specification of that. Furthermore, the corollary mentioned at the end of Sect. 4:

RC: Procedure values can be passed as parameters without any restriction.

covers a majority of practically relevant cases not involving lifetime guarantees. Finally, the concept of lifetime guarantees, which is introduced in full generality in [3] to avoid any unnecessary conceptual restrictions, is usually only needed for the special and simple case `OF MODULE` (restricting the values of a procedure variable to global procedures), which is equivalent to the original rule R0.

References

1. K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language* (Third Edition). Addison-Wesley, Boston, 2000.
2. T.M. Breuel: “Lexical Closures for C++.” In: *Proc. USENIX C++ Technical Conferenc* (Denver, CO, October 1988).
3. C. Heinlein: *Safely Extending Procedure Types to Allow Nested Procedures as Values.* Nr. 2003-03, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, June 2003. `http://www.informatik.uni-ulm.de/pw/berichte`
4. B.W. Kernighan, D.M. Ritchie: *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1988.
5. B. Meyer: *Eiffel: The Language.* Prentice-Hall, New York, 1994.
6. H. Mössenböck, N. Wirth: “The Programming Language Oberon-2.” *Structured Programming* 12 (4) 1991, 179–195.
7. B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.
8. S. Thompson: *Haskell. The Craft of Functional Programming.* Addison-Wesley, Harlow, England, 1996.
9. N. Wirth: *Programming in Modula-2* (Fourth Edition). Springer-Verlag, Berlin, 1988.
10. N. Wirth: “The Programming Language Oberon.” *Software – Practice and Experience* 18 (7) July 1988, 671–690.

Leveraging Managed Frameworks from Modular Languages

K. John Gough and Diane Corney

Queensland University of Technology, Box 2434, Brisbane 4001, Australia
{j.gough,d.corney}@qut.edu.au

Abstract. Managed execution frameworks, such as the *.NET Common Language Runtime* or the *Java Virtual Machine*, provide a rich environment for the creation of application programs. These execution environments are ideally suited for languages that depend on type-safety and the declarative control of feature access. Furthermore, such frameworks typically provide a rich collection of library primitives specialized for almost every domain of application programming. Thus, when a new language is implemented on one of these frameworks it becomes necessary to provide some kind of mapping from the new language to the libraries of the framework. The design of such mappings is challenging since the type-system of the new language may not span the domain exposed in the library application programming interfaces (*API*s).

The nature of these design considerations was clarified in the implementation of the *Gardens Point Component Pascal* (**gpcp**) compiler. In this paper we describe the issues, and the solutions that we settled on in this case. The problems that were solved have a wider applicability than just our example, since they arise whenever any similar language is hosted in such an environment.

Keywords: modular languages, managed execution, compilers, .NET common language runtime, java virtual machine.

1 Introduction

1.1 Managed Execution Frameworks

Managed Execution Frameworks are those computing environments in which the execution is mediated by a runtime, usually a *just in time compiler* (*JIT*). Characteristic of such frameworks is that each program carries a rich store of metadata with the code. This metadata allows the runtime to ensure the type-correctness of the program, and conformance to the declarative access restrictions for each datum. This is the sense in which such frameworks are *managed*.

In every case managed execution frameworks provide facilities for an underlying object-oriented (*OO*) type system. In the case of the *JVM* [2] this type system is sufficient for the implementation of the Java [1] language. In the case of .NET the underlying type system is intended to support a wider range of languages, and is semantically richer [5].

Such frameworks, with their facilities for enforcing type correctness and modularity, are an ideal environment upon which to implement modular languages. Indeed, they are capable of supplying stronger guarantees of integrity than any native execution system, particularly in a component based system.

A further attraction of such frameworks is the richness of the base class libraries that come with the system. These libraries cover almost every possible area of need in the application programming domain. Libraries exist for handling *URLs*, *XML*, cryptography, socket programming, regular expressions, and hundreds more special requirements, as well as all the classic collection classes.

Thus we have a challenge: managed frameworks provide an excellent platform on which to deliver the benefits of specialist languages. However, such excellence will count for nothing, in terms of user productivity, unless the language provides easy access to the treasures of the libraries.

1.2 Gardens Point Component Pascal

Gardens Point Component Pascal (**gpcp**) is a compiler for the Component Pascal language [3]. The language is, in essence, an extension of Oberon-2 [4]. **gpcp** is an open-source implementation of the language. It runs on either the .NET Common Language Runtime (*CLR*) or the *JVM*. There are some slight semantic restrictions on the *JVM* version that have been discussed elsewhere [8]. Nevertheless, the compiler may be self-bootstrapped or cross-boostrapped on either platform.

Implementing a language such as Component Pascal for either of our example managed frameworks poses some interesting challenges. Elsewhere, we have dealt with the mapping of Pascal-family languages to such frameworks [7,9]. In this paper we consider the different issues that are involved in accessing the full domain of the framework libraries from languages with simpler, or simply different, type systems.

In the remainder of the paper we take our experience with **gpcp** as a motivating example of the general issues that are involved when attempting such a mapping.

For both example platforms **gpcp** provides access to the rich library facilities, allowing seamless integration with other languages hosted on the same framework. In each case, simple browser tools are provided that display the library *API*s in (a version of) the syntax of Component Pascal.

1.3 Type System Mismatches

Providing access to the libraries of the host environments posed some critical design problems for **gpcp**.

Whenever a new language is implemented on a managed execution framework it is necessary to map the type system of the new language onto the underlying type system of the platform. In almost all cases the mapping will involve some kind of mismatch. Either some semantic feature of the new language will have no counterpart on the platform, or some semantic feature of the platform type-system will have no counterpart in the new language. Perhaps both.

The type system of Component Pascal is simple, clean and orthogonal, but does not match the underlying type system of either managed execution engine. For example, there is no provision on either platform for arrays of statically declared length: array bounds checks are always performed against the dynamically determined length of each array. Conversely, there is no counterpart in Component Pascal for the *protected* access mode for object members on the type systems of both platforms.

The type-system omissions do not prevent the implementation of the full semantics of Component Pascal, as described in the book [9]. However, the lack of expressivity creates problems for full library access. Thus, the first decision was whether or not to be satisfied with access to a subset of the library facilities, leaving out all those that exposed type-system features that had no Component Pascal counterpart. It was our judgement that such a choice would have been too severe. For example, in order to access the graphical user interface facilities of the libraries it is necessary to be able to participate in the event-handling model of the underlying platform.

1.4 Matching Type-Systems and Harm Minimization

As a matter of philosophy, it was our wish to provide access to the facilities of the base class libraries without doing violence to the simplicity of our target language. We were happy to make the *compiler* much more complex, provided that the type-system as seen by the programmer was effectively unchanged. Thus in some cases the compiler needs to understand the semantics of some "foreign" type-system feature, so as to provide for type-safe access to some library facility. Typically the compiler does not allow programs written in Component Pascal to *define* such features. This principle leads to some subtle phenomena.

The rest of the paper is organized as follows – In the next section we discuss the extensions to the type system of **gpcp** that allow access to the underlying framework libraries. The following sections describe the problems that arise from the mismatch of name-scopes and lexical semantics, and the particular issues that arise in connection with object constructors.

As a matter of notational consistency we refer to the characteristics of the the underlying managed execution engine as the *framework* attributes. The characteristics of the language for which the compiler is created are called the *source* attributes.

For the most part we take our examples from the .NET platform, which has the richer of the two underlying type-systems. We occasionally refer to some significant differences with the *JVM*.

2 Extending the Type System

2.1 Features That Do *NOT* Need Mapping

There are a number of framework type system features that do not impact on the user of the libraries. In effect, features that do not appear in framework-*API*s may be ignored.

Anonymous Classes from Java. Anonymous classes cannot appear in the visible part of an framework-*API*, and hence may be ignored for purposes of library use. Curiously, the underlying *JVM* execution platform knows nothing of anonymous classes either. Such classes are not unnamed, it is just that their automatically generated names are not legal Java identifiers.

Initialize-Only Fields in .NET. Initialize-only fields are only able to be set from within a value constructor. Although there is no syntax to define such fields in Component Pascal, their presence is not problematic. Such fields are simply imported, read-only fields to the user. Furthermore, such fields have their immutable status guaranteed by the runtime system.

Properties in .NET. *Properties*, in C#, are methods of objects that are accessed by means of field-access syntax. Properties may have a *setter* method, a *getter* method or both. The methods may simply access an underlying, private field, or may produce their values by an arbitrary computation. In any case, such properties are simply syntactic sugar encapsulating calls to conventionally named methods. For example, the apparent field access – "`stream.Position`" – in C# translates into a call to the method –

```
call instance int64 System.IO.FileStream::get_Position()
```

Such properties appear in the framework-*API* as instance methods that may be called from source languages that do not have the property notion.

Static Methods and Fields. Ordinary procedures and variables in Component Pascal source are mapped into static methods and static fields in the framework type-system. These static members belong to a dummy "module-body" class that has no instance data. Conversely, record types with type-bound methods in our source language map onto the framework type-system as classes that have instance fields and methods, but have no static members. Classes in the framework in general have both instance and static members. Furthermore, the static members have names that are only unique when qualified by the classname. No syntactic extension is required to access such static features in the framework. However, there is an additional semantic demand on the compiler.

In the base language a fully qualified name has the form –

ModuleName.Designator

where the designator itself may be a dotted name. The semantic extension is to recognize that when a designator begins with an identifier that is an imported, framework typename the qualified name is to be understood as –

ModuleName.ClassName.Designator

We did invent a "syntax" for the output of the class browser that represents such static features. An example foreign module that exports a single class with a single static field and method might be displayed as in Fig. 1. In this display format static features are shown textually bound to the class, following the "keyword" *STATIC*.

2.2 Missing Accessibility Modes

The framework-*API*s of the libraries on both platforms expose features with "*protected*" accessibility. Such features are only accessible from within the code of sub-classes of the

```
FOREIGN MODULE Foo;
  TYPE Bar* = POINTER TO RECORD
                (* Instance fields go here *)
              STATIC
                myField* : INTEGER;
                PROCEDURE MyMethod*(a,b : CHAR);
              END;
END Foo.
```

Fig. 1. Display format for static framework features

class that declares the feature. We wish to be able to define types that are extensions of framework types, so protected features must be visible to our source programs. However, if a program attempts to exploit such visibility by making an illegal access it will raise an exception at runtime. In keeping with general principles we must then insist that the compiler trap such illegal accesses at compile time.

This particular example is a test of our stated philosophy. Our compiler must understand the semantics of the protected accessibility, while doing the least damage to the Component Pascal type-system. As a consequence, our implementation has no syntax for *declaring* protected fields or procedures[1], but understands the semantics of framework features that are so marked. The programmer does not even need to understand the foreign semantics. Any illegal accesses are trapped by the compiler, and attract the message – "*This name is only accessible in extensions of the defining type*".

2.3 Implementing Interfaces

A more substantial test is posed by the need to access library features that require interface implementation. Both .NET and *JVM* platforms have type-systems that allow for classes to extend a single super-type, but to implement multiple interfaces (fully abstract classes). Declaring that a type implements a particular interface places an obligation on the declarer to provide methods to implement each of the abstract methods of the interface. Many of the methods of the *API*s have formal parameters of interface type.

Firstly, it should be noted that it is technically possible to avoid the need to declare Component Pascal types that implement interfaces. We may use a wrapper technique. The idea is as follows – a type is defined in the wrapper, written (say) in C#, that declares that it implements the needed interface type. The wrapper must fulfill its obligations by providing (possibly abstract) implementations for all methods of the interface type. A Component Pascal program may then extend the wrapper type, and provide concrete implementations for any abstract interface methods. Unfortunately, this plausible attempt is insufficient. A compiler that does not understand the semantics of interfaces will reject type-assertions that "cast" an actual parameter of the wrapper type to the formal, foreign interface type. Of course, the type-cast could also be moved to yet another wrapper, but the outlook is not promising and this approach was quickly rejected.

[1] As a matter of practicability the *class browser* has a distinctive mark for such features in foreign modules, but the mark cannot be used in program texts.

gpcp provides a wrapper-free mechanism for implementing interface types from the framework. This requires a syntax extension to declare that a Component Pascal type promises such an implementation. Here is an example –

```
TYPE Foo = POINTER TO RECORD (A + B + C) ... END
```

The type *Foo* extends the type *A* and implements interfaces *B* and *C*. In the case that the *A* is left out, the type extends the universal supertype *ANYPTR*.

The compiler needs not only to recognize the extended syntax, but also to enforce the semantic obligations of such a declaration. A significant modification is necessary for the semantic check that all inherited abstract methods have been implemented. As well as exploring a linear sequence of super-types, a directed acyclic graph of super-interfaces may also need to be explored.

In keeping with our general philosophy **gpcp** allows and supports access to interface types defined in other languages. However, there is no provision to allow definition of such types within the base language.

2.4 Events in .NET

Event handling in .NET is based on objects that extend the *System.MulticastDelegate* type. Delegates are a generalization of type-safe procedure types (function pointers). The .NET version of **gpcp** uses the delegate mechanism of the *CLR* to implement Component Pascal's procedure types.

Delegates extend procedure types by allowing delegate values to be bound to a particular object instance at the time of the value creation. Any subsequent call of the delegate value dispatches the encapsulated (instance) method with the encapsulated object instance as receiver. Functional programmers may usefully think of this mechanism as currying the first argument of the call.

GUI programming on either target platform requires participation in the respective event handling model. In the case of .NET, this requires that delegate values be able to be constructed with and without encapsulated object instances. It also requires some kind of syntactic mechanism to attach and detach delegate values to specific event sources. One of the references [9], gives a detailed discussion of the alternatives to language extension. For **gpcp** we decided to avoid the rather arcane contortions of the reference and add a small syntax extension.

Event types are declared with the same syntax as procedure type declarations. For example –

```
TYPE Foo = EVENT(ch : CHAR);
```

declares that *Foo* is an event type that can hold a list of callbacks to procedures with a single, value-mode, formal parameter of character type. In this case `EVENT` is a new keyword.

Values of delegate type may be created and attached to an event source using syntax that specifies: the source, of some event type; the callback procedure, of some matching procedure type, and optionally nominates an object instance on which the callback is to be dispatched. The syntactic form relies on a builtin procedure, the semantics of which are thus known to the compiler. The procedure call is of the form –

`REGISTER(`*event, callback*`);`

where *event* is a designator denoting an event variable. The actual parameter designator *callback* may simply denote a conformant procedure name. Alternatively, and more usually, the callback will be of the form *object.procedure*. In this case the designator *object* denotes the target object instance, and *procedure* denotes a conformant procedure that is type-bound to the type of the object. There is a corresponding `DEREGISTER` builtin procedure, that detaches the callback from the event source.

For this particular extension, we allow Component Pascal programs to define their own event types, as well as being able to use imported, foreign types.

3 Name Scopes and Lexical Issues

3.1 Identifier Syntax

Identifiers in our frameworks-lexicon, are less constrained than is the case for typical source programming languages. This is an advantage in some contexts, since it allows a compiler to generate synthetic identifiers that are guaranteed not to clash with any source identifier.

From the point of view of a library *user* the only really significant danger is that some framework identifier might clash with a reserved word in our source language. This is easily resolved by using a suitable escape mechanism. In the case of **gpcp** this is the backquote character, "‘".

3.2 Overloaded Names

When names are matched within the execution framework two names are taken to be the same name only if the complete name-and-signature string is the same. This is a nicely simple rule, but from the point of view of a *user* the appearance is one of name-overloading. For example a library may define any number of methods with the same simple name, provided they have different call-signatures. From the point of view of the user of the library these are a set of methods with overloaded names. This poses a particular problem for languages that do not allow overloaded names.

There are a number of different approaches to resolve this problem, and we experimented with three different approaches during the development of **gpcp**.

Firstly, it is possible for the user, or the compiler vendor, to provide a mapping of overloaded names in the framework libraries to unique (and mnemonic) names in the user namespace. In **gpcp** we implemented a mechanism in which "dummy definition modules" declare the exported features of a particular library. As part of that definition the name by which the feature is to be known in Component Pascal is also declared. Here is a typical snippet –

```
FOREIGN MODULE Console;
  PROCEDURE WriteI*["Write"](i : INTEGER);
  PROCEDURE WriteR*["Write"](r : REAL);
  PROCEDURE WriteC*["Write"](c : CHAR);
  ...
```

The literal string in brackets declares the platform identifier to which the unique names are to be mapped in the object file.

The difficulty with such a scheme is that name maps need to be manually produced for all the libraries. Such a scheme does not seem sensible except perhaps for languages that have a well-organized user community and an authoratative central repository for all mappings.

The problem with manually produced name maps is that they are unpredictable, so that different programmers producing maps for the same library will produce different maps, hindering portability. What however if the mappings are algorithmically produced?

We experimented with automatic name-mapping for **gpcp** for some time, but such schemes have other disadvantages. Methods were given names that were based on the (overloaded) platform name together with a suffix computed from the formal parameters. For methods with formal parameters of named object types the full names became too long. In such cases the algorithm denoted the type of the formal parameter by a single character code, and appended a five-digit hashcode suffix to the name. This scheme produced deterministic names, but the names had no mnemonic content and thus obfuscated the user code.

Finally, we decided that directly exposing the overloaded names to the programmer was the least of the evils[2]. We nevertheless attempted to limit the cognitive impact of introducing this foreign concept into programs. This change necessitated a relatively major modification to the **gpcp** source. Component Pascal has a number of features that are syntactically ambiguous. For example, consider the situation when a designator is being parsed and a left parenthesis follows an identifier –

```
Foo.Bar( ...
```

It is unclear whether what is expected next is an actual parameter list, or a type-test. The original design of **gpcp** bound all identifiers "on the fly" and used semantic information to determine whether *Bar* in the fragment is a procedure name, or denotes a dynamically typed variable. The obvious problem with such a design is that if *Bar* denotes an overloaded method in the current scope, then the name cannot be bound until the actual parameter types are known. In any case, the techniques for handling overloaded names are a well known part of compiler folk-lore. It is just that if we had known we were going to add these semantics later we would have started off with a different parser architecture.

The final version of the compiler accepts overloaded names for framework methods. Since such a feature is alien to the Pascal family of languages, the matching algorithm that we implemented is deliberately conservative. When a source program name is bound, as a first step a *list* is made of all matching methods. This initial match takes into account all automatic coercions of basic types, and any allowed sub-type polymorphism. If the list has length one, all is well. If the list has length zero the usual *identifier not known* error is notified. Finally, if the list is of length more than one an exact binding is attempted on each list element. Exact binding does not take into account *any* coercions. If the second

[2] Independently, the implementers of Fujitsu's COBOL.NET compiler followed the same path and implemented overloading in that compiler.

binding matches exactly one list element the binding succeeds, but a warning is issued to the user –

```
 106     str := JL.String.valueOf(ENTIER(num * 10.0 + 0.5));
**** ---------------------^
**** Warning: Multiple overloaded procedure signatures match this call
**** Bound to     - valueOf(LONGINT)
**** Matches with - valueOf(REAL)
**** Matches with - valueOf(SHORTREAL)
```

In this example the exact binding is displayed, and the alternative bindings that require coercions are listed also.

If the second binding attempt produces no exact match then the program is rejected and an error notified to the user. The error message lists all of the possible matches. The user has the option of changing the program so as to select the method that was intended, by making it an exact match.

The use of a conservative policy in name binding is well justified. Most languages that allow overloading of identifiers define rules by which ambiguity is to be resolved. It is assumed (perhaps erroneously) that the programmer will be familiar with the detailed "tie-breaking" rules that the particular language uses. The situation for us is quite different. We cannot assume that the programmer will understand the rules, since the whole construct is alien to the base language.

One final point requires elaboration. In general, Component Pascal programs may *refer* to overloaded identifiers, but may not *define* their own overloaded identifiers. However, a subtle but necessary exception must be made. Component Pascal programs must be able to override extensible methods with overloaded names. Thus, if a Component Pascal module needs to override a method that has an overloaded name, the new method must be *defined* with the same overloaded name as the method that it replaces.

4 Constructors

4.1 Default Constructors

Component Pascal, along with other members of its family, uses a builtin procedure `NEW()` to allocate object instances. The object code produced for this construct invariably allocates some memory from the heap, and then executes some initialization code. If the class of the new object is an extension of some other class the initialization code begins by recursively calling the so-called "no-args constructor" of the super-class. This ensures that any private fields of the super-class are correctly initialized. Notice that there is no syntactic mechanism for specializing the initialization. If an object instance requires further initialization this must be specified outside of the constructor.

The object creation semantics of Component Pascal thus corresponds to the default constructor behaviour of languages such as C# and Java.

Unfortunately this mechanism is insufficient for interoperation with most *OO* frameworks. The problem is that in such frameworks there is the possibility that correct initialization of objects of some type *requires* the specification of initial values. For such types it is mandatory to supply arguments to each constructor call. This poses an immediate problem, since we have no syntax to call such constructors for imported foreign types. Furthermore, since we have no syntax for *specifying* constructor bodies we have no location in which to describe how the construction of an object of an extended type should recursively call the "with-args constructor" of the super-type. We experimented with a number of different syntactic extensions for **gpcp** before finding a design that seemed sufficiently expressive.

As a necessary first step, the semantic analysis of the compiler had to be extended so that it detected cases where foreign classes do not have a public no-arg constructor. In such cases any attempt to call `NEW()` on a variable of the type or any derived type must be rejected as an error.

4.2 Calling Constructors with Arguments

The **gpcp** class browser presents the meta-data associated with framework constructors in a way that helps the user remember how to call the constructors in the source language. For example, a constructor for the class *M.Rectangle*, taking two integer arguments, might appear as in Fig. 2. The constructor is represented as a value-returning function

```
FOREIGN MODULE M;
  TYPE Rectangle* =
    POINTER TO RECORD
      ...
    STATIC
      PROCEDURE Init*(w,h : INTEGER) : Rectangle,CONSTRUCTOR;
    END;
END M.
```

Fig. 2. Display format for with-args constructor

that is called as if it were a static method of the class. This constructor would be typically called using a statement of the form –

```
newRec := M.Rectangle.Init(10,15);
```

The use of the *CONSTRUCTOR* marker[3] in the signature declaration deserves some explanation. The use of such markers is part of the Component Pascal style, but is not strictly necessary in the class-browser application. It is true that the *compiler* needs to know that this is a constructor, and not just an ordinary value returning function. However the *user* does not really need to know. The reason that the marker is displayed here is that it presages the use of the same mechanism in the declarative context that is discussed in the next section. In the declarative context the marker is mandatory.

[3] For the technically inclined this identifier is *not* a reserved word, but is rather a *context sensitive mark*. The token only has meaning in this syntactic context, and does not clash with other uses of the token as an identifier.

4.3 Specifying Constructors with Arguments

Suppose now that we wish to define types that are extensions of framework types, where those framework types do not have public no-args constructors. In order to achieve this we need to be able to define constructors in Component Pascal that will specify how the with-args supertype constructor is to be recursively invoked. This requires a substantial syntactic extension.

Constructors with arguments are rather strange creatures. Constructors have two purposes: to cause the allocation of some memory, and to initialize the state of the new object. These two purposes are syntactically indivisible in most source languages, and cannot be separated in verifiable code on either of our target platforms. This poses an interesting language design issue: how is one to specify the recursive invocation of the super-type "constructor"? Clearly we need only the second part of the functionality, since the memory allocation has already been made. Various languages have adopted various tricks for doing this. Our solution is most similar to that of C#, which we think has a coherent approach.

The declaration of an explicit constructor in Component Pascal follows the syntactic pattern in Fig. 3. In this syntax the uppercase identifiers are all literals. Identifiers in

```
PROCEDURE procName '(' Formals ')' ':' typeName ',' BASE ',' '(' Actuals ')' ';'
  LocalDeclarations
BEGIN
  Statements
END procName ';' .
```

Fig. 3. Syntax for constructor with arguments

italics are lexical and syntactic categories which have their usual meanings. *Actuals* is a comma-separated list of expressions with names bound in the scope of the formal parameter list.

The effect of such a declaration is understood by the compiler as follows. The presence of the context sensitive mark *base* indicates that this is a constructor. The type to which the constructor is bound is determined by the return-type of the procedure.

The code generated by the compiler is determined by the statement sequence in the procedure body in the normal way. However, the code of the body is preceeded by a prolog that recursively calls one of the constructors of the super-type (that is, the base type). The parameter values that are passed to the recursive call are computed by evaluating the expressions specified in "*Actuals*" in Fig. 3. The choice of super-type constructor is determined by the static types of the actual parameter expressions, using the overload resolution machinery described in Sect. 3.2.

An additional, semantic point relates to the manner in which the body of the constructor may refer to its own newly allocated object. We treat all constructors as though they begin with a "magic" declaration of a local variable of their characteristic type, named *SELF*. It is a semantic constraint that every reachable path through the control flow of the constructor body must end in the statement –

```
RETURN SELF
```

as might be expected, **gpcp** checks this constraint as part of its data-flow analysis.

As a final point we note that a simplified syntax is available for one common case. In this case only the procedure header of the constructor needs to be specified. The formal parameters are simply passed to the super-type constructor with an equal signature. The effect of the simple declaration –

```
PROCEDURE Init(a,b : INTEGER) : MyType,CONSTRUCTOR;
```

is exactly equivalent to the following, degenerate example in the full syntax –

```
PROCEDURE Init(a,b : INTEGER) : MyType,BASE(a,b);
BEGIN RETURN SELF END Init;
```

5 Conclusions and Future Issues

Our experience with **gpcp** suggests that it is possible for languages such as Component Pascal to have full access to the facilities of the libraries of the popular managed execution platforms. We believe that we have managed to do this without adding significantly to the base language.

The base language accepted by **gpcp** is relatively unchanged. Nothing has been left out, and the additions that have been made may only be used when they *must* be used. Furthermore, for the most part the additional semantics of the platform type system may be ignored by the programmer. It has been our goal to ensure that the error messages associated with incorrect use of framework semantics explain clearly what the required semantics are. The example given earlier for *protected* features is typical.

The price that has to be paid for safe access to the libraries is a very substantial increase in the complexity of the compiler. The compiler must not only represent the semantics of its own language, Component Pascal in our case, but must also represent most of the semantics of the framework type-system. We think this is a good bargain: compilers are complicated exactly so that they can make programming safer and more productive.

In fact, the most troublesome area in the adaptation of the compiler arose from a seemingly trivial detail. The graph of the "imports" relation in Component Pascal modules is always acyclic. The original version of **gpcp** depended in a fundamental way on this guarantee. No such property holds for the packages of the Java libraries, or the assemblies of the .NET system, so major changes to the type-resolution mechanisms had to be made.

In this discussion we have omitted mention of exception handling. On both target platforms library methods may throw exceptions under circumstances where allowing the program to abort would be unreasonable. We added a minimal extension to allow such exceptions to be caught. This involves both syntactic and sematic changes that are discussed elsewhere [9].

Looking toward the future, it seems likely that future versions of the .NET *CLR* will support parametric polymorphism (generics). Direct support for such a type-system extension will allow powerful additions to the standard libraries.

At the very least compilers targetting the *CLR* will need to be able to recognize the metadata associated with generics. Going further, we think that it will be useful to provide users of **gpcp** access to the features of generic libraries. The best way to do this is a subject of ongoing debate in the development group.

References

1. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading MA, 1997.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA, 1997.
3. Oberon Microsystems, 'Component Pascal Language Report' available from – `http://www.oberon.ch/resources`
4. H Mössenböck and N. Wirth, 'The Programming Language Oberon-2'; *Structured Programming* 12, 179–195.
5. Microsoft, '.NET Common Language Infrastructure'. Also as ECMA standard ECMA-335, available from – `http://www.ecma.ch/publications/standards/ecma-335`
6. Microsoft, 'C# Language'. Also as ECMA standard ECMA-334, available from – `http://www.ecma.ch/publications/standards/ecma-335`
7. K John Gough and Diane Corney, 'Evaluating the Java Virtual Machine as a Target for Languages other than Java', *Joint Modula Languages Conference*, Zurich 2000.
8. K. John Gough, 'Stacking them up: A Comparison of Virtual Machines'; *Australian Computer Systems and Architecture Conference*, Gold Coast, Australia, February 2001. Also available from – `http://www.citi.qut.edu.au/research/plas/projects/cp_files`
9. John Gough, 'Compiling for the .NET Common Language Runtime', Prentice-Hall PTR, Upper Saddle River, New Jersey, 2002.

Systems = Components + Languages: Building a Flexible Real-Time Simulation and Test Environment

Brian Kirk, Stuart Doyle, Toby Harris, and Tom Priestley

Robinson Associates, Weavers House, Friday Street
Painswick, Gloucestershire, GL6 6QJ, UK
enquires@robinsons.co.uk
http://www.robinsons.co.uk

Abstract. Building a flexible system is difficult, particularly if each instance of the system will be highly customised. This paper describes how an ERTMS Railway Simulator and Test Environment system was designed using abstractions, components and purpose specific languages. The use of components and modularisation provides the basis for reliability and testability, whereas the use of languages provides for flexibility to match the generic simulation environment to the problem in hand. The challenge of using Commercial Off The Shelf (COTS) hardware and software for large real-time systems is also discussed.

1 Introduction

There is a Chinese curse 'may you live in exciting times', we are undoubtedly living in exciting times. As a commercial Software House we generally find that projects have become more vaguely defined, results are demanded in impossible time scales and budgets are limited. It was ever thus, but now it is ever more so. Somehow, we are told, we must provide 'off-the-peg' solutions and deliver a series of evolving working systems which help to clear the specification 'fog' that seems to be the hallmark of modern business. This paper describes the pragmatic approach to building systems, which we have developed using techniques similar to those of *extreme programming* [Extreme], but applied in a systematic way. Most importantly it has enabled us to conserve our engineering integrity and dignity in the face of current commercial realities. It also provides a system level model for building large flexible real-time systems. The practical example system we use is a railway simulation and system test environment, it provides a nice example to illustrate the techniques used and lessons learned. To our surprise, purpose specific languages (PSLs) have become a crucial tool for providing productivity and flexibility in the application and testing of systems. These 'bespoke' languages are used to illustrate how different kinds of flexibility are provided within the overall system. In this case one is used to describe the static aspects of the railway (track layout, etc) and others are used to orchestrate the system behaviour during simulation of railway operation in real-time.

Early in the development Python was used as the system implementation language and development environment [Python]. As the project developed and more concurrency was introduced Python was augmented with specific languages developed using CoCo [ReMo], this improved both simplicity and system performance.

2 The Problem

Background

In the past the railway signalling system in each country has been produced to National Standards. As a result there is little compatibility between systems across Europe, it is estimated that a train going from the north-west tip to the south-east tip of Europe would need two carriages dedicated just to signalling equipment! Furthermore the existing systems work on the traffic light principle, the railway routes being separated into safe segments of track by traffic lights protecting their entry points. During operation railway signallers reserve temporary safe paths through the track network and the traffic light signals indicate which trains can proceed and which must stop in order to avoid collisions. In the new system each train is driven at a safe braking distance behind the train in front according to its local conditions.
In order to harmonise standards and improve traffic throughput Europe the European Railway Transport Management System [ERTMS, EN50128] is being developed. It defines a set of services for controlling trains safely and a means to access them.

Practical Issues

Commuters are by nature patient people; they have little choice. But even the most patient of them would object to their railway system being shut down for ‘system testing’ of a new signalling system! Clearly it’s not economical to build huge test tracks (it takes about 10 km to stop a high-speed train safely). So simulation is used to

1. Verify that the new ERTMS and ETCS concepts will work (expensive if wrong!)
2. Verify that the new system will work for the track network topologies and layouts that exist or which are being (re)designed
3. Verify the system performance of timetables of train operation, before investing huge amounts of capital expenditure required to make them feasible.
4. Try out the interfaces between existing (legacy) systems with ERTMS and ETCS.
5. Test that new pieces of equipment function and perform correctly *in real time*.

The main practical problem is that a generic simulator is needed which can be easily configured for a given track layout and the set of operational scenes that it is intended to support. As we shall see bespoke languages have been developed to naturally express each of these needs. The general approach to the design is based on software engineering best practice [Cool], to provide the desired flexibility of reconfiguration..

System Abstraction

Before a problem can be solved there needs to be some kind of ‘pivot’ which is shared between the problem and its solution. In this case we chose as the pivot an abstraction of the overall railway system, this is illustrated in Fig. 1.

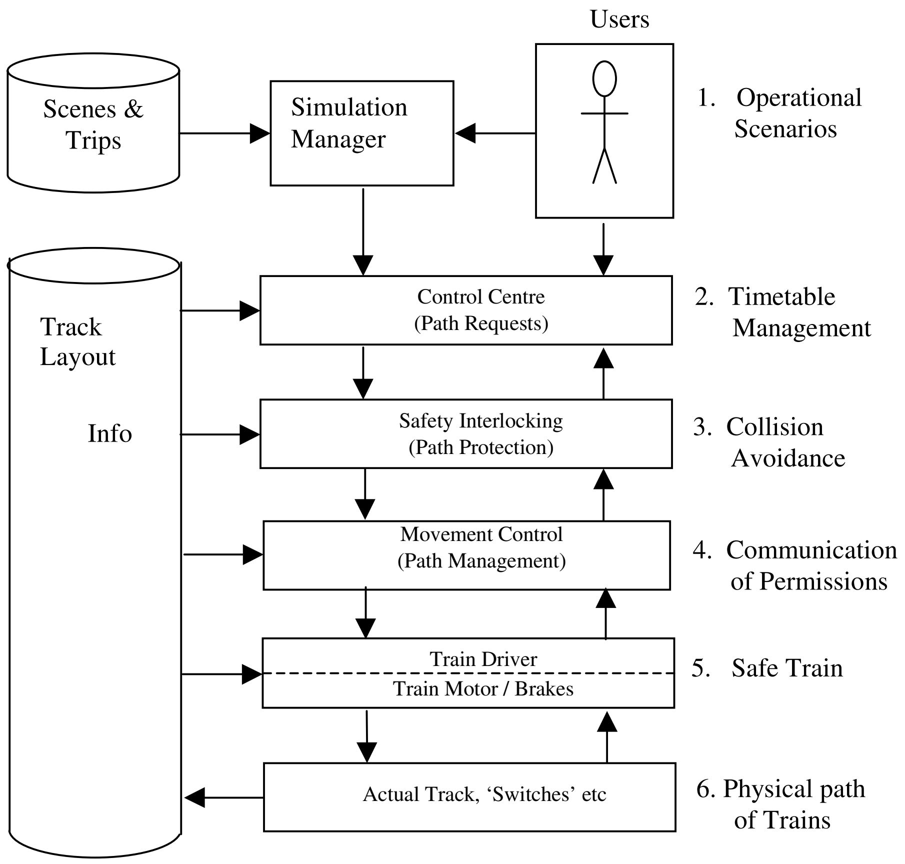

Users
Scenes & Trips
Simulation Manager
1. Operational Scenarios
Track Layout Info
Control Centre (Path Requests)
2. Timetable Management
Safety Interlocking (Path Protection)
3. Collision Avoidance
Movement Control (Path Management)
4. Communication of Permissions
Train Driver
Train Motor / Brakes
5. Safe Train
Actual Track, ‘Switches’ etc
6. Physical path of Trains

Fig. 1. Levels Of system abstraction

There are six aspects of the system that have been abstracted, thus partitioning the system at a top level on the basis of ‘separation of concerns’. The six aspects are

1. *Operational scenarios* which define the intended operational outcome of the railway system. Within this there are two sets of users. They are manual users (e.g. Operators in a real railway) or a simulation manager (computer) which replays predefined scenarios, which describe the intended operations. This provides for unattended operation of the simulations, trials of disaster scenarios and repeated runs to assess rail network performance and regression testing support for changes.

2. *Timetable management* which basically is concerned with requesting, reserving and displaying safe pathways through the rail network for a succession of train journeys. Energy management *across* the system may also be important.
3. *Collision avoidance* which uses conventional 'interlocking' rule sets to ensure that active pathways through the rail network are used in a mutually exclusive way.
4. *Movement control* which is concerned with indicating to the trains how fast, how far and when they can move. This can be signalled either by traffic lights or by sending 'movement permission' information for use by the driver of the train.
5. *Safe train movement* which involves permitted movement of the train and other functions such as safe rates of acceleration and deceleration (to conserve the passengers' health!) observing speed restrictions and ensuring that trains stop in alignment with platforms etc.
6. Physical environment which includes the actual track, its network map (in three dimensions), level crossings, station platforms, points (switches), track crossovers, track ends etc. This physical reality has to be consistently modelled and the information provided to all the other abstractions. Note that the current topology i.e. network state) of the track may change over time as the direction of points change and trains occupy crossovers or other physically shared pathways. The track network paths dynamically change whenever the points are switched.

System Architecture

The system architecture is shown in Figs. 2 and 3, it follows the general shape of simulation (and test) systems that is illustrated in Fig. 2. In this case Railway Control Centres and Interlocking processors already exist physically and operationally and can be incorporated into the simulator; those are shown in bold in Fig. 3. This concept of a hybrid system, with some real parts and some simulated parts, is crucial for the overall system to function both as a system simulator and/or a test environment for new subsystems.

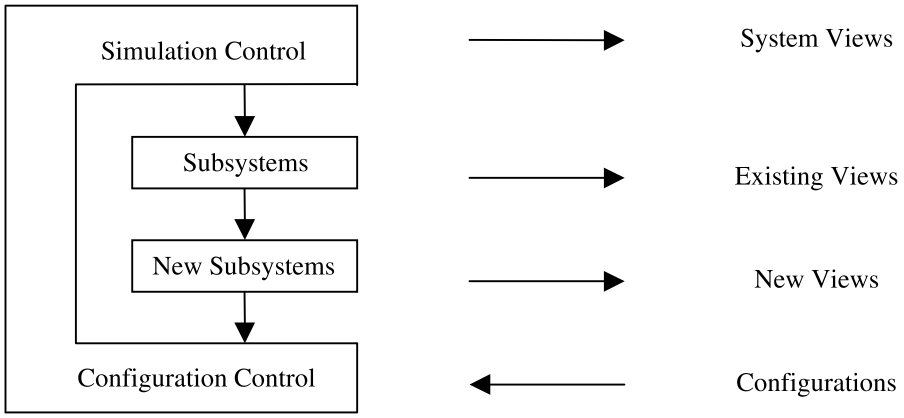


Fig. 2. Architectural concept

Abstractions to Architecture

The system works as follows (please refer to Fig. 2 above and Fig. 3 below).

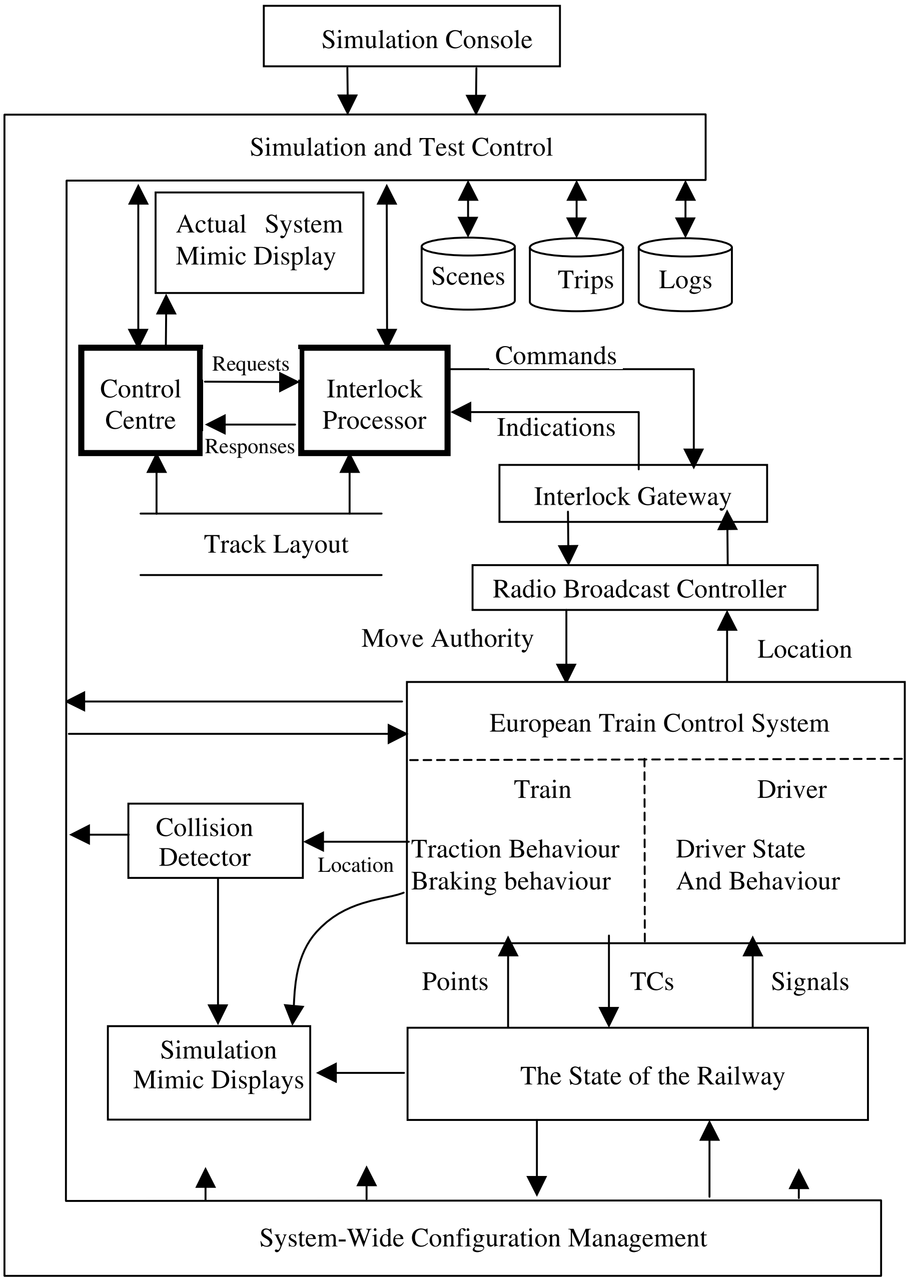


Fig. 3. System architecture and design

Before a simulation is run it has to be defined, this entails

1. Defining the purpose of the simulation and the expected outcomes.
2. Defining the track layout network on which the trains will travel.
3. For each train defining its trip through the network, locations visited and time to be taken. A set of such trips is a timetable.
4. Defining the initial state of the railway (position of points etc), initial placement of the trains, event triggers for data logging etc. This information is called a scene.
5. Defining which information should be logged as the simulation proceeds

As we shall see later various languages were developed to clearly, separately and meaningfully express those aspects of the simulation definition.

Next the abstractions defined in Figure 1 were mapped into a system architecture, whilst keeping in mind a multi-processor object oriented system design for the implementation [KiNiPu]. The result is shown in Figure 3, the main features are

1. Levels of abstraction are highest at the top of the page.
2. The simulation and test control overarches the whole system and in turn is driven by the scenes, trips and logging which characterise a particular simulation run.
3. Control Centre and Interlocking which in our case can be the actual equipment or simulators of it. The Control Centre produces its own mimic diagrams (aimed at Railway Managers and Traffic Controllers). The new subsystems follow, starting with the Interlocking Gateway. This is in effect system glue which matches the internals of the new subsystems with the existing data formats and protocols of actual Interlocking Processors(IPs). This is necessary as there are several types and generations of Interlocking Processors. (going right back to relay logic on many current railways), so this gateway provides for mapping to legacy systems.
4. The Radio Broadcast Controller can either be simulated or actual, in fact it was being tested during the development of the simulator. It takes the commands from the IP and translates them into radio packets which are (safely) broadcast to the trains from track side beacons (balises) which are located in the track layout.
5. Each train has three main functions. The ETCS system provides system interlocking and overall safe control of the train's movement from location to location. Beneath that there is a train controller which manages the trip lifecycle in terms of starting and stopping the train in a desirable way (safe, efficient) and also a driver controller which either supports an automated or human driver.
6. As the simulation proceeds the State of the Railway is updated by the objects in the subsystems. In turn other subsystems of objects, such as each train, refer to the State of the Railway as it defines their context within the simulation environment.
7. The Collision Detector continuously (and independently) monitors the traffic flow.
8. Finally there is a sophisticated Mimic Display which provides representations of various levels of abstraction for the track layout, trains drivers, collision risk etc.

3 Implementing the Scripting Language

Using a scripting language to provide flexibility and glue together the various components of the system proved to be very effective, particularly as instances of real railway systems are so varied in their layout, train characteristics etc. So several scripting languages were devised based on the abstractions in Figure 1.

Kinds of Description Languages

So far the following Purpose Specific Languages have been defined

1. Layout Description Language
This defines the track layout and placement of all physical objects at static locations in the real world. The object types are stratified onto various conceptual layers related to the physicality of the railway (e.g. track, signals, balises …) to support flexible rendering of mimic diagrams
2. Trip Description Language
This defines each (dynamic) journey that each train instance takes through the rail network.
3. Scene Description Language
This defines a particular scenario that must be simulated i.e. the initial conditions and ongoing characteristics of a simulation e.g. event triggers at given locations for data capture.
4. Interface Description Language

This defines the mapping between real world equipment (data formats, commands, and protocols) and the internal world of the simulator. Its purpose is to avoid as much as possible any detailed redesign or reprogramming of the simulator itself when incorporating a new Interlocking, Control Centre or other real subsystem into the overall simulation and system.

Producing the Language Compilers

Having done some prototyping in Python it was decided to go for higher performance based on a multiprocessor concurrent simulator implementation. It was clear that a language compiler would produce faster runtimes than the Python interpreter by a factor of about 10. The CoCo [ReMo] compiler compiler was chosen as a tool for producing the compilers and as the rest of the system was already written in Borland Delphi (Object Pascal) a CoCo implementation [CoCo] was evaluated based on Delphi. After some teething problems, our largest production 'broke' the CoCo implementation, the implementer tuned the CoCo implementation. Once this was repaired we had no further compiler problems at all. The CoCo system provides:

1. A language for precisely and concisely specifying the syntax of the target language
2. A means for embedding semantic actions at various points in the grammar, thence referenced as an Attributed Grammar
3. A parser generator and set of class libraries which takes the Attributed Grammar, processes it and produces the source code of a compiler for the target languages.

As with all such tools getting the first language defined and processed is tedious, thereafter it becomes a 'handle turning' process for subsequent languages.

The process of producing each compiler is illustrated in Fig. 4.

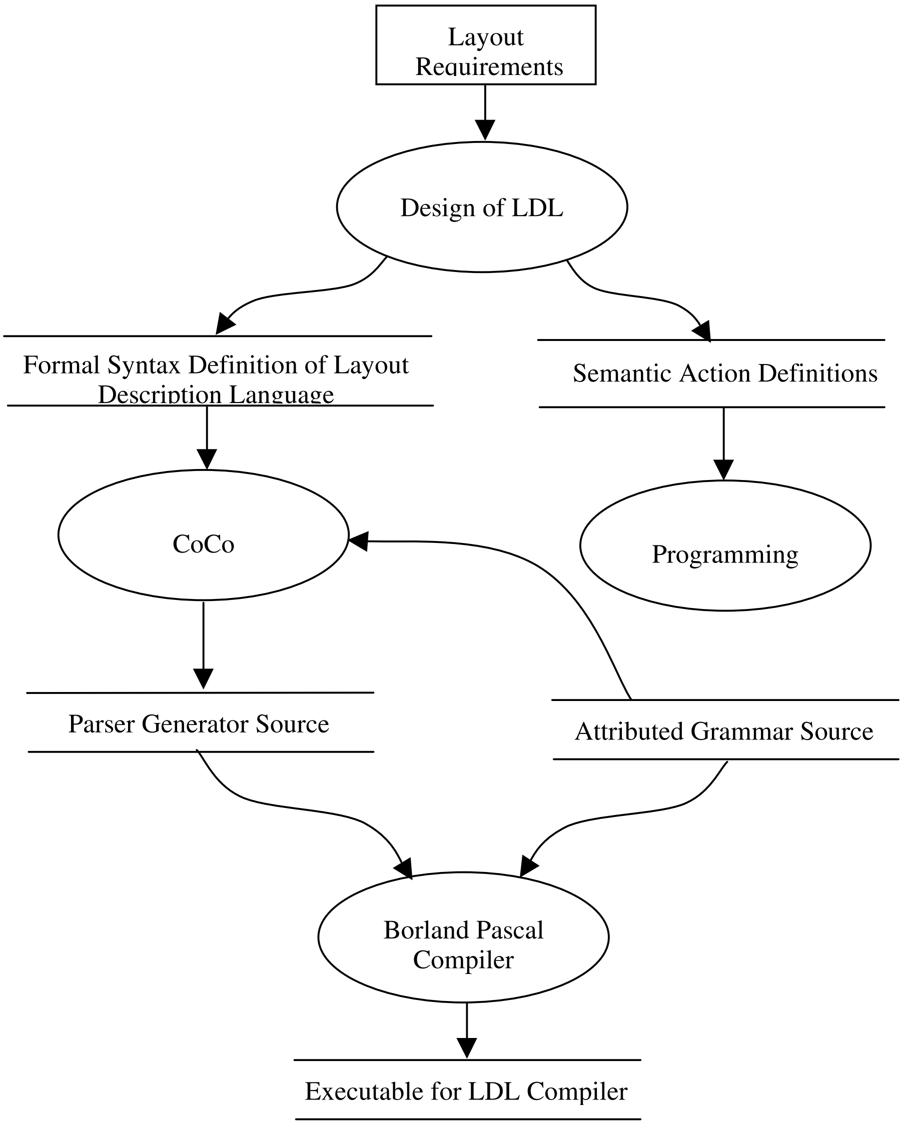

Layout Requirements
Design of LDL
Formal Syntax Definition of Layout Description Language
Semantic Action Definitions
CoCo
Programming
Parser Generator Source
Attributed Grammar Source
Borland Pascal Compiler
Executable for LDL Compiler

Fig. 4. Producing the layout description language (LDL) compiler

Some of the additional benefits of this approach are

1. The language grammar is unambiguous and formally specified
2. The compiler that CoCo produces is effectively a state machine and hence traceability for code verification is straight forward (for safety related systems)
3. CoCo has excellent diagnostics, which are invaluable to fledgeling language designers who don't naturally think like LL(1) parsers!

The end result from CoCo is a compiler class which can then be linked onto the component of the simulator which needs to execute that kind of script.

4 Implementing Concurrency

The challenge here was to base the simulator on a network of standard PCs running a standard operating system (Windows NT). To avoid 're-inventing the wheel' the performance of COM and DCOM [RA] processes and message passing were investigated. The idea here was that if the performance of the basic scheduling and message passing were adequate then the simulator subsystems (and their objects) would be would be mapped onto this commercial off the shelf (COTS) platform. The concurrency model is sketched in Fig. 5.

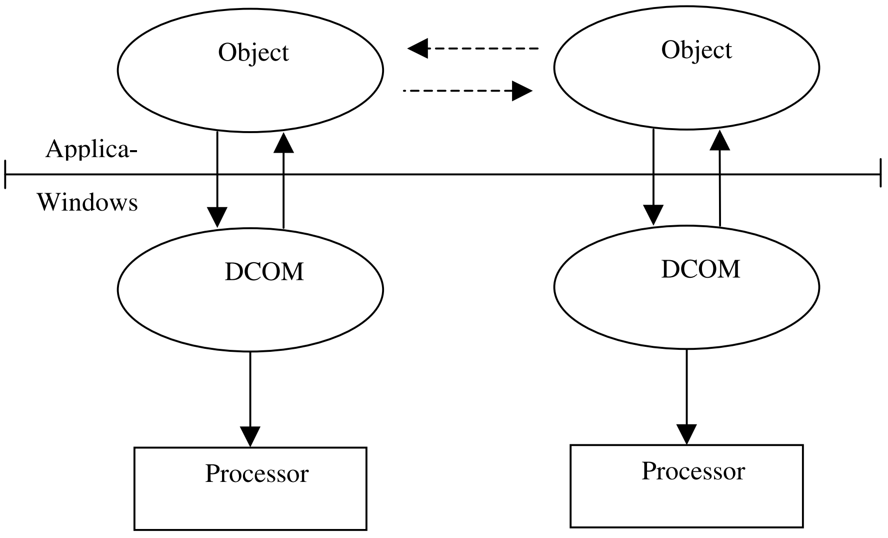


Fig. 5. Transparent (static) distribution of objects to processors and communication between objects

The findings of our evaluation indicated the following message passing performance on a 2.2 GHz PC running Windows NT [RA] with a 100 MHz Ethernet.

Message passing/method invocation Mechanism	**Performance**
Via Method call with parameters(message)	100k/sec
Via COM to COM message, same processor	50k/sec
Via DCOM to DCOM message, same processors	40k/sec
Via DCOM to DCOM message, different processors	25k/sec

Based on the actual real time needs of the simulator for each subsystem the appropriate mechanism was chosen. A further consideration was to minimise the number of ways scheduling can occur (simplification) as long as their performance is comfortably adequate for the purpose. Also the need to provide objects within subsystem with a separation of concurrency if they might become separate physical implementations at a later date. In this case they can be implemented as either a simulation or as a real device as long as their interface is conserved [Parnas].

It is much easier to change the implementation of a particular object if it has a literal message passing interface and its own concurrency. The modularisation (processes, modules etc) here is based on the current and future needs of the project.

An example of this is the train subsystem, which started out life using traditional method calls i.e. a set of cooperating objects sequentially scheduled via method calls (ETCS, train, driver). Later each object in the subsystem was remapped onto a concurrent process so that real ETCS equipment could be connected and also that a real driver interface could be connected to replace the train's simulated driver objects.

5 Conclusions

As the project has developed a number of conclusions have been drawn:

1. The intended end users can now write scripts to customise the system with high productivity, little training is needed as the system now 'speaks their language'.
2. The use of purpose specific languages (PSLs) greatly supports the notion of system design by 'separation of concerns', in turn this greatly benefits the simplicity of the overall system implementation.
3. Compiler construction tools are now robust and easy to use, they have truly moved from the domain of academia to that of system development by competent engineers
4. The formality implicit in defining the purpose, syntax and semantics of the PSLs has brought clarity and discipline to the project. It has also greatly assisted with traceability through the project's documentation, from problem statement to executable system.
5. There is a great benefit in basing the system implementation model on the implicit concurrency of the problem's reality.
6. It is possible to design flexible real time systems using simple techniques and tools, hence: systems = components + languages.

Acknowledgments. This work has been based on many discussions and people's ideas and experience. We would like to thank Prof. H. P. Mössenböck for his excellent CoCo system and Mark Reith of Tetzel Inscriptions for his reliable Delphi implementation of it. Thanks also to Libero Nigro for his collaboration and discussions on the analysis and synthesis of concurrent systems. Brian Kirk would like to thank his colleagues at Robinsons, and also Jim and Niall Cooling in particular, for their support during the past year.

Fortunately the light at the end of the tunnel was not an approaching train!

References

[Alex] Alexander C. Notes on the Synthesis of Form. *Harvard University Press*, Cambridge, Massachusetts, 1964

[CoCo] Coco/R websites are www.tetzel.com, www.scifac.ru.ac.za/coco/, www.scifac.ru.ac.za/compilers/

[Cool] Cooling, J.E. Software for Real Time Systems. ISBN 0-201-59620-2 (pbk), 2002

[EN 50128] Railway applications – Software for Railway Control and Protection Systems *EN 50128*, CENELEC, May 2000

[ERTMS] ERTMS/ETCS System Requirements Specification, Subset 026 Issue 2.2.2

[Extreme] Extreme Programming web site is www.extremeprogramming.org

[KiNiPu] Kirk B, Nigro L and Pupo F: Using Real Time Constraints for Modularisation. *LNCS, 1204,* Springer Verlag, pp. 236–251, 1997

[Parnas] Parnas D. L. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, December, 1972

[Python] Python web site is www.python.org

[RA] Robinson Associates. Performance of COM and DCOM. *Internal Report* October 2002 Email: Enquiries@robinsons.co.uk

[ReMo] Rechenberg, Peter and Hanspeter Mössenböck. A compiler Generator for Microcomputers, Hertfordshire UK, *Prentice Hall* 1989

Event Library: An Object-Oriented Library for Event-Driven Design

Volkan Arslan, Piotr Nienaltowski, and Karine Arnout

Swiss Federal Institute of Technology (ETH), Chair of Software Engineering
8092 Zurich, Switzerland
{Volkan.Arslan,Piotr.Nienaltowski,Karine.Arnout}@inf.ethz.ch
http://se.inf.ethz.ch

Abstract. The Event Library is a simple library that provides a solution to the common problems in event-driven programming. Its careful design facilitates further extensions, in order to satisfy users' advanced needs. It has been implemented in Eiffel, taking advantage of the advanced mechanisms of this language. In this paper, we present the architecture of the library and show how it can be used for building event-driven software. The discussion is illustrated with a sample application.

1 Introduction

Event-driven programming has gained considerable popularity in the software engineering world over the past few years. Several systems based on this approach have been developed, in particular Graphical User Interface (GUI) applications. Event-driven techniques facilitate the separation of concerns: the application layer (also called *business logic*) provides the operations to execute, whereas the GUI layer triggers their execution in response to human users' actions.

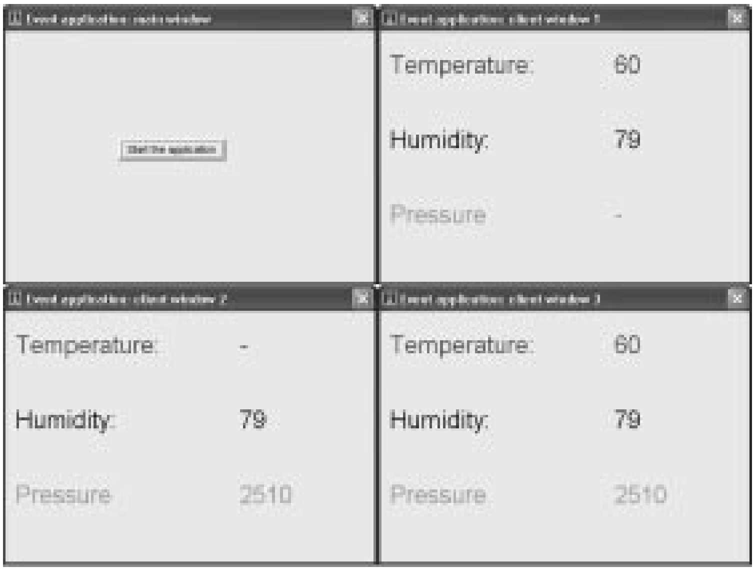
Similar ideas have been proposed under the names *Publish/Subscribe* and *Observer pattern*. The latter, introduced in [6], purports to provide an elegant way to building event-driven systems. It certainly contains useful guidelines for the development of event-driven programs; however, it falls short when talking about reuse: developers have to implement the pattern anew for each application. In order to solve this problem we decided to turn the Observer pattern into an Eiffel library, so that it can be reused without additional programming effort.

Despite its small size — just one class — the Event Library is powerful enough to implement the most common event-driven techniques, and it can be extended to handle users' advanced needs. Its simplicity results from using specific mechanisms of Eiffel, including Design by Contract™, constrained genericity, multiple inheritance, agents, and tuples. The underlying concepts of event-driven programming are presented in [9]. The same article provides also a review of existing techniques, such as the .NET delegate mechanism and "Web Forms". In this paper, we focus on the Event Library.

The rest of the paper is organized in the following way. Section 2 explains how to use the Event Library, and illustrates it with an example. Section 3 describes the architecture of the Event Library and shows how it fulfills the *space*-, *time*-, and *flow-decoupling* requirements of the Publish/Subscribe paradigm. Section 4 draws conclusions and discusses possible extensions of the library.

2 Event Library in Action

We use a sample application to show the basic capabilities of the Event Library. We explain, step by step, how to use the Event Library to build an event-driven application. Both the library and the example presented here are available from [1]. They were developed with the EiffelStudio 5.2 graphical development environment [3], and they can be used on any platform supported by EiffelStudio[1].


Event application: root window
Start the application
Event application: client window 1
Event application: client window 2
Event application: client window 3

```
Temperature:   60
Humidity:      79
Pressure       -

Temperature:   -
Humidity:      79
Pressure       2510

Temperature:   60
Humidity:      79
Pressure       2510
```

Fig. 1. Sample event application

2.1 Sample Event-Driven Application

We want to observe the temperature, humidity, and pressure of containers in a chemical plant. We assume that the measurement is done by external sensors. Whenever the

[1] Microsoft Windows, Linux, Unix, VMS, Solaris

value of one or more measured physical attributes changes, the concerned parts of our system (e.g. display units) should be notified, so that they can update the values.

There are several reasons for choosing an event-driven architecture for such application. First of all, we should take into account the event-driven nature of the problem: input values are coming from external sensors at unpredictable moments, and the application is reacting to their change. Secondly, the independence between the GUI and the "business logic" is preserved. If the physical setup changes (e.g. sensors are replaced by different ones, new display units are introduced), the system can be easily adapted, without the need to rewrite the whole application.

Compiling and launching the sample causes four windows to appear (Fig. 1). The top-left window is the main application window. It displays information about the subsequent execution phases of the application. Three client windows display the values of temperature, humidity, and pressure in a chemical container. Note that these three display units correspond to the same container; however, they are interested in different sets of measures: Client window 1 shows temperature and humidity; Client window 2 shows humidity and pressure; Client window 3 shows temperature, humidity, and pressure. Each of these windows can change its "interests" over time, either by subscribing to a given event type (see 2.5) or by unsubscribing from it (see 2.6). All subscriptions may be also temporarily suspended (see 2.7).

2.2 Using the Event Library

Figure 2 shows the overall architecture of our sample application. BON notation [11] is used. The arrows represent the client-supplier relationship.

The application is divided into three clusters: `event`, `application`, and `gui`. The `event` cluster contains one class, `EVENT_TYPE`, which abstracts the general notion of an *event type*. The `application` cluster contains the application-specific class `SENSOR`. The `gui` cluster groups GUI-related classes, including `MAIN_WINDOW`, `APPLICATION_WINDOW`, and `START`.

Class `SENSOR` models the physical sensor that measures temperature, humidity, and pressure. The class contains the corresponding three attributes: `temperature`, `humidity`, and `pressure`, used for recording values read on the physical sensor.

```
class SENSOR
  feature -- Access
    temperature: INTEGER
          -- Container temperature
    humidity: INTEGER
          -- Container humidity
    pressure: INTEGER
          -- Container pressure
end
```

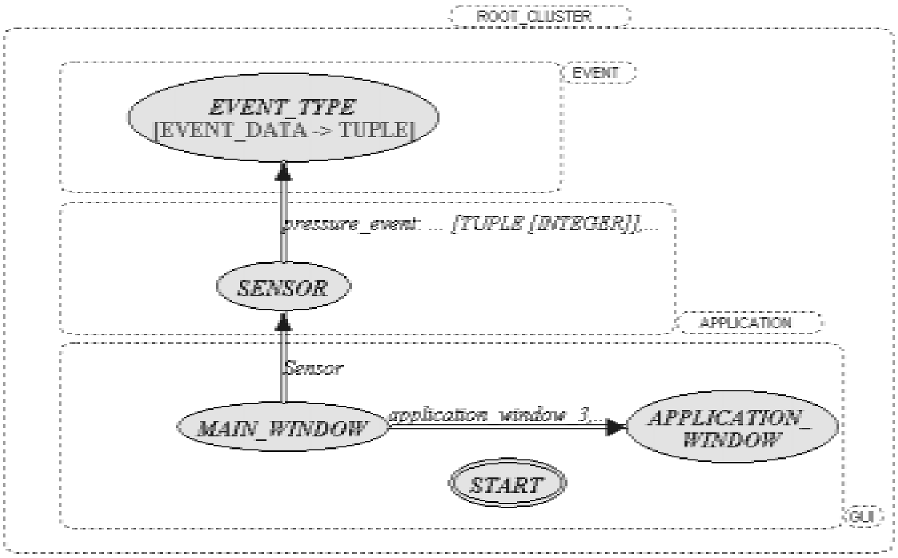


Fig. 2. Class diagram of the sample application

A `set` feature is provided for each attribute, e.g.

```
    set_temperature (a_temperature: INTEGER) is
            -- Set temperature to a_temperature.
      require
        valid_temperature:
          a_temperature > -100 and a_temperature < 1000
      do
        temperature := a_temperature
      ensure
        temperature_set: temperature = a_temperature
      end
```

Note the use of assertions — preconditions and postconditions — to ensure correctness conditions. The *precondition* states that the temperature read by the sensor must be between -100° and 1000°; the *postcondition* ensures that the temperature is equal to the temperature read by the sensor.[2]

2.3 Creating an Event Type

We need to define an event type corresponding to the change of attribute `temperature` in class `SENSOR`; let us call it `temperature_event`:

[2] For the purpose of the subsequent discussion, in particular in code examples, we will only use the attribute `temperature`. Similar code is provided for attributes `humidity` and `pressure`, although it does not appear in the article.

```
feature -- Events
  temperature_event: EVENT_TYPE [TUPLE [INTEGER]]
    -- Event associated with attribute temperature
invariant
  temperature_event_not_void: temperature_event /= Void
```

To define the temperature event, we use the class `EVENT_TYPE` (from the `event` cluster), declared as `EVENT_TYPE [EVENT_DATA -> TUPLE]`. It is a generic class with constrained generic parameter `EVENT_DATA` representing a tuple of arbitrary types. In the case of `temperature_event`, generic parameter is of type `TUPLE [INTEGER]` since the event data (temperature value) is of type `INTEGER`.[3]

2.4 Publishing an Event

After having declared `temperature_event` in class `SENSOR`, we should make sure that the corresponding event is published when the temperature changes. Feature `set_temperature` of class `SENSOR` is extended for this purpose:

```
set_temperature (a_temperature: INTEGER) is
    -- Set temperature to a_temperature.
    -- Publish the change of temperature.
  require
    valid_temperature:
      a_temperature > -100 and a_temperature < 1000
  do
    temperature := a_temperature
    temperature_event.publish ([temperature])
  ensure
    temperature_set: temperature = a_temperature
  end
```

The extension consists in calling `publish` with argument `[temperature]` (corresponding to the new temperature value) on `temperature_event`. Class `SENSOR` is the *publisher* of the temperature event.

2.5 Subscribing to an Event Type

We need to complete our sample application with other classes that will subscribe to events published by `SENSOR`. First, we introduce class `APPLICATION_WINDOW` in the `gui` cluster with three features `display_temperature`, `display_humidity`, and `display_pressure`. `APPLICATION_WINDOW` is a *subscribed* class: it reacts to the published events by executing the corresponding routine(s), e.g. `display_temperature`.

[3] This definition complies with the constraint `EVENT_DATA -> TUPLE` since `TUPLE [INTEGER]` conforms to `TUPLE` [4].

Secondly, we introduce class MAIN_WINDOW, which is in charge of subscribing the three features of class APPLICATION_WINDOW listed above to the corresponding three event types (temperature_event, humidity_event, and pressure_event). In order to subscribe feature display_temperature of application_window_1 to event type temperature_event, the subscriber makes the following call:

```
Sensor.temperature_event.subscribe
 (agent application_window_1.display_temperature (?))
```

As a result, feature display_temperature of application_window_1 will be called each time temperature_event is published. The actual argument of feature subscribe in class EVENT_TYPE is an agent expression[4]: **agent** application_window_1.display_temperature(?). The question mark is an open argument that will be filled with concrete event data (value of type INTEGER) when feature display_temperature is executed [2].
Let's have a closer look at feature subscribe of class EVENT_TYPE:

```
subscribe (an_action: PROCEDURE [ANY, EVENT_DATA])
         -- Add an_action to the subscription list.
  require
    an_action_not_void: an_action /= Void
    an_action_not_already_subscribed:
         not has(an_action)
  ensure
    an_action_added:
         count = old count + 1 and has (an_action)
    index_at_same_position: index = old index
```

subscribe takes an argument of type PROCEDURE [ANY, EVENT_DATA].[5] The first formal generic parameter (of type ANY) represents the base type on which the procedure will be called; the second formal generic parameter (of type EVENT_DATA, which is derived from TUPLE), represents the open arguments of the procedure. This procedure will be called when the event is published. It has to be non-void and not already among listed actions, as stated by the precondition of subscribe. This means that the same procedure cannot be subscribed more than once to the same event type. The postcondition ensures that the list of subscribed actions is correctly updated.

2.6 Unsubscribing from an Event Type

Class EVENT_TYPE provides feature *unsubscribe*, which allows objects subscribed to an event type to cancel their subscription. Feature start_actions of class

[4] **agent** x.f(a) is an object representing the operation x.f(a). Such objects, called agents, are used in Eiffel to "wrap" routine calls [2]. One can think of agents as a more sophisticated form of .NET delegates.

[5] This argument is an *agent*.

`MAIN_WINDOW` uses it to unsubscribe `application_window_1` from event type `temperature_event`:

```
Sensor.temperature_event.unsubscribe
  (agent application_window_1.display_temperature)
```

The implementation of `unsubscribe` is similar to that of `subscribe`; it just does the opposite: unsubscribes the procedure from the event type.

2.7 Additional Features of Class EVENT_TYPE

Besides procedures `subscribe`, `unsubscribe`, and `publish` that we have already seen, class `EVENT_TYPE` has three additional features: `suspend_subscription`, `restore_subscription`, and `is_suspended`. It is possible to define custom event types by inheriting from `EVENT_TYPE` and redefining or adding specific features. This is explained in detail in [9].

3 Architecture of the Event Library

In this section, we discuss the architecture of the Event Library. We also show how the library fulfills the requirement of *space-*, *time-*, and *flow-decoupling* of the event-driven paradigm.

3.1 Basic Concepts

The design of the library relies on a few basic concepts: *event type*, *event*, *publisher*, *subscriber*, and *subscribed object*. Let us have a closer look at these notions.

3.1.1 Events and Event Types

The concepts of event and event type are often confused. To reason about event-driven design, in particular in the object-oriented setting, one should understand the difference between them.

In event-driven systems, the interaction between different parts of the application and external actors (such as users, mechanical devices, sensors) is usually based on a data structure called the *event-action table*. This data structure records what action should be executed by some part of the system in response to each *event* caused either by another part of the system, or by an external actor. Thus an *event* is a signal: it represents the occurrence of some action taken by the user (e.g. clicking a button) or a state change of some other parts of the system (e.g. temperature change measured by the sensor). An *event type* provides the abstraction for *events*. In other words, an event is an instance of the corresponding event type. For example, every time the user clicks a mouse button, an event of event type `Mouse_Click` is published. In our sample application, each temperature change caused an event of type `temperature_event` to be published.

3.1.2 Publishers, Subscribers, and Subscribed Objects

Publisher is the part of the system capable of triggering events. The action of triggering an event is called publishing. In the Event Library, this is achieved by calling feature *publish* of the corresponding EVENT_TYPE object.

Subscribed objects are notified whenever an event of the corresponding event type is published. The notification is done by calling the feature of the subscribed object previously registered within the event type. In the Event Library, the *agent mechanism* is used for registration and notification of subscribed objects. An object may be subscribed to several event types at a time, and receive notification from all of them. Conversely, an event type may have several subscribed objects.

Subscriber is in charge of registering subscribed objects to a given event type. In the Event Library, this is achieved by calling feature subscribe of the corresponding EVENT_TYPE object. We introduce a separation between the concepts of subscriber and subscribed object. It is important to note that such distinction provides another level of abstraction in the event-driven design, although in most cases subscribed objects are their own subscribers.

3.2 Implementation

Class EVENT_TYPE inherits from the generic class LINKED_LIST from the EiffelBase Library [8]. Therefore all features of LINKED_LIST are available to the class EVENT_TYPE. In a previous version of the Event Library, client-supplier relation was used instead of inheritance. We opted for the inheritance-based solution because it is an easy way to implement the subscription list (class EVENT_TYPE can itself be considered as a list of subscribed agents); it also facilitates future extensions of the library, e.g. through the redefinition of features in class EVENT_TYPE. On the other hand, it introduces a potential risk: some features inherited from the class LINKED_LIST might be misused by the clients of EVENT_TYPE, e.g. a client having access to an event type could simply clear the list. Therefore, one may want to hide these features by changing their export status to {NONE}, thus preventing the clients from using them directly.

3.3 Space-, Flow-, and Time-Decoupling

Event-driven architectures may provide separation of application layers in three dimensions: *space*, *time*, and *flow*. Such decoupling increases scalability by removing all explicit dependencies between the interacting participants. [5] provides a short survey of traditional interaction paradigms like message passing, RPC, notifications, shared memory spaces, and message queuing; they all fail to provide time, space and flow decoupling at the same time. It is interesting to note that, in spite of its simplicity and small size, the Event Library can provide decoupling in all three dimensions.

Space Decoupling. The publisher and the subscribed class do not know each other. There is no relationship (neither client-supplier nor inheritance) between the classes SENSOR and APPLICATION_WINDOW in Fig. 2. Publishers and subscribed classes are absolutely independent of each other: publishers have no references to the subscribed objects, nor do they know how many subscribed objects participate in the interaction. Conversely, subscribed objects do not have any references to the publishers, nor do they know how many publishers are involved into the interaction. In our sample application, class SENSOR has a client-supplier relationship to the class EVENT_TYPE, but no relationship to subscribed classes. Only class MAIN_WINDOW, which is the *subscriber*, knows the publisher and the subscribed objects. In fact, in the general case, the subscriber does not have to know the publisher; it has to know the event type to which subscribe an action. Event types are declared in the publisher class SENSOR; this is why class MAIN_WINDOW has a reference to the class SENSOR. Had the event types been declared in another class, the subscriber class would keep no reference to the publisher. On the other hand, declaring the event type outside the scope of the publisher (SENSOR) might be dangerous: every client having access to the event type (e.g temperature_event) can publish new events. In such case, the publisher (SENSOR) would have no possibility to control the use of that event type.

Flow Decoupling. Publishers should not be blocked while publishing events. Conversely, subscribed objects should be able to get notified about an event while performing other actions; they should not need to "pull" actively for events. Obviously, support for concurrent execution is necessary in order to achieve flow decoupling. In our sample application, there is no such support; therefore flow decoupling does not exist: publisher objects are blocked until all subscribed objects are notified.

The Event Library can ensure flow decoupling, provided that *publisher*, *event type* and *subscribed* objects are handled by independent threads of control, e.g. SCOOP *processors* [7, 10]. See Sect. 4 for more details.

Time Decoupling. Publishers and subscribed objects do not have to participate actively in the interaction at the same time. Such property is particularly useful in a distributed setting, where publishers and subscribed objects may get disconnected, e.g. due to network problems.

Current implementation of the Event Library does not provide time decoupling. Nevertheless, it can be easily extended to cover this requirement. The basis for such extension should be, as in the case of flow decoupling, a support for concurrent and distributed execution (see Sect. 4).

4 Current Limitations and Future Work

Initially, our goal was to provide a reusable library that implements the *Observer pattern*. We soon realized that the Event Library can be turned into something much more powerful: a simple and easy to use library for event-driven programming. Despite its small size, it caters for most event-based applications. Whenever more advanced features are needed, the library can be easily extended.

An important contribution of our approach is the distinction between the concepts of *subscribed* and *subscriber* objects (see 3.1). Such separation of concepts brings an additional level of abstraction in application design, thus facilitating reasoning about event-driven systems.

Future enhancements of the Event Library could include "conditional event subscription" for subscribed objects only interested in events fulfilling certain criteria or conditions. For example, objects subscribed to event type `temperature_event` may want to be notified of a temperature change only if the value of attribute `temperature` is between 25 and 50 degrees.

A support for concurrent and distributed execution is another important extension of the library. In particular, flow- and time-decoupling cannot be provided without such support (see 3.3). We plan to base the extension on the SCOOP model [7]. SCOOP provides high-level mechanisms for concurrent and distributed object-oriented programming. We are currently implementing the model for Microsoft .NET [10]; other platforms (POSIX threads, MPI) are also targeted.

Acknowledgements. We would like to thank Bernd Schoeller, Susanne Cech, and Vaidas Gasiunas, for their comments. Bertrand Meyer contributed essentially to the design of the library. We also benefited from the very helpful support from Julian Rogers from ESI.

References

1. Arslan V.: Event Library, at http://se.inf.ethz.ch/people/arslan/
2. Eiffel Software Inc.: Agents, iteration and introspection, at http://archive.eiffel.com/doc/manuals/language/agent/agent.pdf
3. Eiffel Software Inc.: EiffelStudio 5.2, at http://www.eiffel.com.
4. Eiffel Software Inc.: Tuples, at http://archive.eiffel.com/doc/manuals/language/tuples/page.html
5. Eugster P. Th., Felber P., Guerraoui R., Kermarrec A.-M.: The Many Faces of Publish/Subscribe, Technical Report 200104 at http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200104.pdf.
6. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edition, Addison-Wesley, 1995.
7. Meyer B.: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.
8. Meyer B.: Reusable Software: The Base Object-Oriented Component Libraries, Prentice Hall, 1994.
9. Meyer B.: The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design, at http://www.inf.ethz.ch/~meyer/ongoing/events.pdf.
10. Nienaltowski P., Arslan V.: SCOOPLI: a library for concurrent object-oriented programming on .NET, in Proceedings of the 1st International Workshop on C# and .NET Technologies, University of West Bohemia, 5-8 February 2003, Pilsen, Czech Republic.
11. Walden K., Nerson J.-M.: Seamless object-oriented software architecture, Prentice Hall, 1995.

Controlling Access to Distributed Object Frameworks

Karl Blümlinger, Christof Dallermassl, Heimo Haub, and Philipp Zambelli

Institute for Information Processing and Computer Supported New Media
Graz, University of Technology, Austria
{kbluem,cdaller,hhaub,pzamb}@iicm.edu

Abstract. This paper describes the interface between an application and the kernel module of the Dinopolis distributed object framework. It shows the advantages of using dynamic proxies and why special care has to be taken for objects that are passed from the application to the core. Further, a mechanisms to allow the distributed system to identify the issuer of the call is shown.

1 Introduction

The Dinopolis distributed object framework is an open source, Java framework that allows to build highly distributed applications. Its modular design is described in detail in [1]. This paper concentrates on the interface between applications that use the Dinopolis framework and the Dinopolis core modules (also called the Dinopolis kernel). It describes the usage of Java dynamic proxies for this purpose and discusses when they are needed.

As in every network based system, security (authentication and authorization) plays an important role. This paper describes the basic mechanisms, how Dinopolis applications communicate with the Dinopolis kernel and how these calls may be intercepted to implement security checks, account information checks, logging, etc. and how the Dinopolis kernel retrieves the information it needs to perform these checks using dynamic proxies and execution contexts. A long version of this paper is available at
`http://www.dinopolis.org/publications/`

2 Motivation

The internal architecture of the Dinopolis distributed object system is similar to an operating system: Different layers are responsible for different functionality in the system [2]. For the mechanisms described in this paper, it is enough to distinguish between the two major layers: The *user layer* where applications that use the distributed object system are situated and the *kernel layer* where the core functionality of the distributed system is located.

To enforce security policies, the kernel must be able to allow or deny access to kernel objects or to part of the object's methods.

It has to be able to intercept all calls from the user space to any objects in the kernel space, to identify the entity (user or application) that issued the call, and to identify the target object of the call. The last requirement is easily fulfilled in object oriented programming languages, as the target is equal to the object a method is invoked on. So the main problems are to *break up the execution chain* and to *identify the issuer* of the call.

3 (Dynamic) Proxies

According to Gamma [3] a proxy provides "a surrogate or placeholder for another object to control access to it". This definition fits perfectly the need to intercept calls traversing the line from user space to kernel space. If objects that were created in the kernel space are wrapped into a proxy before they are handed to the application, all method invocations on these proxies could be intercepted and forwarded to the security-, logging-, accounting-, or any other kernel module before potentially executed on the target object.

The idea to use a proxy for this purpose is not new at all [4]. But using a static proxy has a couple of severe drawbacks. The creation of the proxy classes could be done with a pre-processor or manually, but whichever method is used, it has to be done at compile time! In highly dynamic distributed object systems, classes may be used that are not known at compile time (of the local system). In this case, the remote system needs to provide the proxy class as well. The enforcement of the local security policy would therefore rely on the proxy implementation of a (potentially unreliable) remote system. This is a big security leak and completely unacceptable.

The solution to this problem is the use of *Dynamic Proxies*. A dynamic proxy is created at runtime and can basically mimic any interface(s). It is therefore perfect for the purpose of wrapping (almost) arbitrary objects that are passed from kernel space to user space. The gain of flexibility of dynamic proxies compensates the loss of performance (slower execution and higher memory usage) in the Dinopolis system.

Another disadvantage of dynamic proxies in Java is the fact that a dynamic proxy cannot be created for an existing (abstract) class, but only for one or more interfaces. This influences the design of the Application Programming Interface (API) of the kernel space, as all objects passed need to be defined by an interface.

4 Execution Context

The possibility to intercept a call from the application to the kernel space is absolutely necessary, but without the possibility to identify the principal (individual, corporation, role, or any other entity), a security check or logging is either impossible or not very useful.

The kernel access layer has to identify the issuer that invoked the method on a kernel object. The term we used for this piece of information is *execution context.*

Different possibilities exist for realizing execution contexts: Thread based, ticket based, or proxy based. The proxy based approach has many advantages compared to the others and was therefore chosen in the Dinopolis system. It is easy to use, does not change the API, and is thread-independent (and therefore also works for Java GUI-events seamlessly).

5 Realisation

Section 3 outlined the dynamic proxy technology to intercept any calls that pass the border between user space and kernel space. The kernel has to wrap any objects, that are returned from a method call before they are passed to user space, into a proxy, so the security checks may not be circumvented by the use of this return-object. The wrapping is necessary for all objects that provide access to the kernel space. If any kernel objects would be passed without being wrapped, the application could invoke a method without the possibility to intercept the call by the security manager or similar modules.

For identification of the issuer it is necessary to copy the execution context to this newly created dynamic proxy (resp. its invocation handler). So when the object is finally returned to the requesting application, any method calls on it may be intercepted and the caller can be identified by the execution context.

Most arguments that are passed to methods of kernel objects have to be treated in a special way: If the argument is a wrapped kernel object (that was previously passed from the kernel to the user space), the kernel access layer has to unwrap the kernel object from the proxy and forward this object to the kernel. Otherwise, the argument is some other object provided by the application (e.g. an observer for some kind of kernel events). This case is just as dangerous as the other way around (but not so obvious): When the kernel invokes a method on this object, an unchecked call bypasses the kernel interception mechanism. Any arguments in this method call cannot be checked and therefore any kernel objects that are passed as arguments are not wrapped into a proxy. This results in an unwrapped kernel object that may be accessed from the application and therefore circumvents the kernel's security mechanisms.

6 Related Work

Various papers describe the possibility to intercept method calls by inserting additional code into existing classes, by changing the classloader, the Virtual Machine or by using a preprocessor [5,6,7]. All of them insert code in all instances of the given class and therefore every single call to these objects is intercepted, even calls inside the kernel which is not wanted.

Sun's Java Authentication and Authorization Service (JAAS) attaches the execution context information to the current thread by letting the programmer wrap every action into a special object [8]. This is very uncomfortable for application programmers.

CORBA and Microsoft's .NET Framework use a token based system to identify the principal. This token is passed on to the target system as an element in the communication protocol [9,10]. As the Dinopolis distributed object system needs authentication and authorization on local systems as well, the protocol based approach cannot be applied in the Dinopolis system. Sun's Remote Method Invocation (RMI) does not implement a security model at all. Authentication and authorization are delegated to applications above the RMI layer (e.g. JAAS) [11].

7 Conclusion

We presented the usage of dynamic proxies as the base element of the interface between an application and the Dinopolis distributed object framework. Dynamic proxies offer the flexibility to intercept calls from user space to kernel space even for classes that are not even known at compile time to perform security checks or other activities. Further, we compared different ways to identify the issuer of a kernel call and how proxies may be used for this purpose as well.

References

1. Klaus Schmaranz: Dinopolis – A Massively Distributable Componentware System, Habilitation Thesis, June (2002)
2. E. Dijkstra: The Structure of the 'THE' Multiprogramming System Communications of ACM, Vol. 11, No. 3, pp. 341–346, May (1968)
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley (1995)
4. Roy H. Campbell, Nayeem Islam, David Raila, and Peer Madeany: Designing and implementing Choices: An object-oriented system in C++. Communications of the ACM, 36(9): 117–126, September (1993)
5. Ian Welch and Robert J. Stroud: Kava - A reflective Java based on Bytecode Rewriting
6. Éric Tanter, Nourny M.N. Bouraqadi-Saâdani and Jacques Noyé: Reflex - Towards an Open Reflective Extension of Java, REFLECTION 2001, Springer Verlag, LNCS 2129, pp. 25–43 (2001)
7. Michael Golm and Jürgen Kleinöder: metaXa and the Future of Reflection, Presented at the OOPSLA Workshop on Reflective Programming in C++ and Java, October 18, 1998, Vancouver, British Columbia
8. Sun: Java Authentication and Authorization Service (JAAS), available online `http://java.sun.com/j2se/1.4/docs/guide/security/jaas/JAASRefGuide.html` (2002)
9. Object Management Group (OMG): Common Object Request Broker Architecture: Core Specification, Version 3.0.2, December 2002
10. J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy: Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication, Microsoft
11. Sun: Java Remote Method Invocation Specification, Revision 1.8, JDK 1.4, available online at `ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf` (2002)

Architectural Aspects of a Thread-Safe Graphical Component System Based on Aos

Thomas M. Frey

Institut für Computersysteme, ETH Zürich
CH-8092 Zürich
frey@inf.ethz.ch
http://www.bluebottle.ethz.ch

Abstract. A message sequencer and an optimized locking mechanism for a multi-threaded graphical component system are proposed as a strategy to minimize locking overhead and code complexity. The strategy allows for fast inter-component communication via delegates in common and simple cases, while still maintaining thread-safety for more complex scenarios. The proposed strategy is applied to a novel graphical user interface framework on the basis of the *Aos* [1] kernel, developed at the ETH Zürich.

1 Introduction

Component systems can be divided into two classes *single-threaded* and *multi-threaded*. Single-threaded software components are generally substantially easier to write and use than multi-threaded implementations because of the requirement for thread safety. For example, the single-threaded case basically does not require any use of locks for inter-component communication. For the few cases where the handling of asynchronous events (generated by input devices, timers etc.) and messages is inevitable, such systems generally offer a message queue that is polled by the main thread of the component hierarchy. Java Swing or .Net WinForms are typical examples of single-threaded graphical component systems.

In a multi-threaded context, several synchronization issues make writing thread-safe components arduous and cumbersome:

State Protection. Most obvious of all, the state of a component must be protected against simultaneous changes by multiple clients. Doing this for a given component is quite simple, although the issues of self-deadlock must be considered. Common ways to avoid self-deadlock are the use of recursive locks or a strict separation of synchronized interface methods from unprotected internal implementation methods [2]. In systems featuring extensible components, great care must be exercised that all extensions of a component adhere to the same chosen locking scheme. Depending on the programming language used, the compiler can help with ensuring the locking discipline. Some research is currently being done in the area of extended static [3] or dynamic [4] checking of code with respect to correctness in terms of a given locking scheme.

Protecting each component's state with an individual lock substantially increases the danger of deadlock, in particular in combination with cyclic component structures.

High-Level Data Races within Components. Another important problem with multi-threaded components are high-level data races. Several independant parameters of a component may have a different semantic meaning when used in combination. For example, consider a graphical panel component that offers thread-safe methods for reading the coordinates of its left and right border. Unfortunately, these methods do not help much when it comes to calculating the width of the panel, since the panel could have been repositioned in the time between the two method calls. There are mainly two strategies to avoid such high-level data races: either offer an explicit lock for the entire component (transaction model) or offer thread-safe accessor methods for all needed combinations of parameters. Detecting high-level data races is a current research topic [5].

High-Level Data Races on an Inter-component Level. Another requirement especially for graphical component systems is the ability to get a consistent view of a group of thread-safe components, for example while rendering, no component may be resized, repositioned, removed from or added to a component hierarchy. The common solution to this problem is to use a hierarchy lock that must be acquired before performing such changes. Without taking great care of locking order, this can be a further source of deadlock.

In the following section we shall derive a solution of compromise that tries to combine the advantages (ease-of-use and efficiency) of single-threaded systems with the advantage of allowing multiple threads cuncurrently calling event listeners.

2 Graphical Component System for Aos

The Aos kernel is a runtime environment based on the metaphor of active objects. Active objects comprise data fields, methods and an activity. Mechanisms for mutual exclusion and synchronization are also provided by the kernel and the *Active Oberon* programming language. Since each activity runs in its own thread, a thread-safe component system was considered desirable.

Similar to most single-threaded graphical component systems, Aos uses a thread and a queue for the handling handling of asynchronous events and a hierarchy lock that ensures a consistent view on inter-component relations. In Aos, the hierarchy lock, the message queue and the thread are combined in an (active) *message sequencer* object. (Fig. 1)

Strict parental control is used as a strategy for drawing and passing of keyboard and mouse events. Other inter-component event-handling is realized by delegate procedures, a combination of a method pointer and object reference. A delegate procedure allows direct component-to-component calls, without the

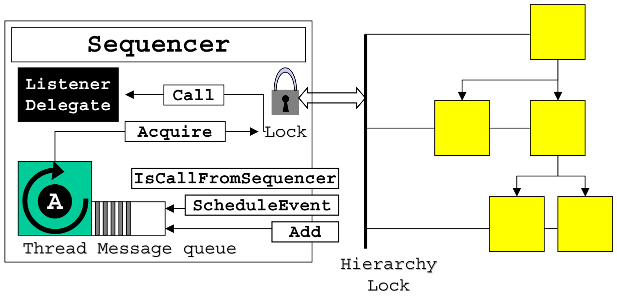


Fig. 1. Sequencer

need of sending a message up and down an entire hierarchy. The delegate procedures allow simple component wiring by registering event listeners at event sources without implementing special interfaces. In a graphical component system, a an inter-component message can be caused by a button being pressed or a string in an editor being changed.

2.1 Synchronization Strategy

In the graphical component system presented, the following strategy for data protection is applied: If an event listener method of an object is called, it first checks if the call originates (directly or indirectly) from the sequencer thread. In this case, no special care is taken, the event consumer just does whatever it needs to do to handle the event. Otherwise, if the method call originates from some other thread, the event-listener re-schedules the event in its associated sequencer object and returns. It will be called again, this time by the sequencer thread, and handle the event. The sequencer thread always acquires the hierarchy lock before calling a component method. Thanks to this strategy, an event handler can always safely operate on its state, no matter from which thread it was originally called. The following code sequence shows a generic event handler:

```
PROCEDURE EventHandler*(sender, data : OBJECT);
BEGIN
  IF ~sequencer.IsCallFromSequencer() THEN (* sync needed ? *)
    sequencer.ScheduleEvent(SELF.EventHandler, sender, data)
  ELSE (* actual business logic *)
  END
END EventHandler;
```

Property methods that change a visual component may only be called by instances holding the respective hierarchy lock. This invariant is ensured by assertions.

As a fine point we note that checking if the current procedure call emanates from the associated sequencer object can be done very efficiently without involving locks. It is done by storing the sequencer's thread object in a private field when it is created and comparing this field to the current thread object. The current thread object can be accessed efficiently via the thread's stack [1].

2.2 Constructing a Component Hierarchy from an XML Description

It was decided that components should be described persistently in XML form. As a consequence, an XML parser [6] was developed. The parser is designed to call specific object generator methods for each tag it encounters. A generic XML object is created, if for a certain tag, no appropriate generator method is found. The generic XML object can store all attributes and sub-elements for later use. The generic XML object also offers a basic query mechanism for finding sub-elements and retrieving attributes. The component system directly uses this mechanism for internalizing XML descriptions, i.e., for building up hierarchical component structures in the memory. Unlike in DOM systems, no separate intermediate XML representation is needed. The wiring of participating components makes use of a path mechanism for locating components after the entire document is loaded.

3 Conclusion

The proposed active sequencer object in combination with an event synchronization strategy is a versatile and efficient means for managing synchronization and locking concerns in a multi-threaded graphical component system. It is not only used in the XML oriented graphical component framework mentioned in this paper but also in the desk-space management subsystem of Aos. The use of delegate procedure variables combined with the protection of event-listener methods (Sect. 2.1) considerably simplify the wiring of components.

References

1. Muller, P. J.: The Active Object System – Design and Multiprocessor Implementation, PhD Thesis, ETH Zürich, Switzerland, 2002
2. Schmidt, D. C.: Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components, C++ Report, vol. 11, Sept. 1999.
3. v. Praun, C., Gross, T.: Static Conflict Analysis for Multi-Threaded Object-Oriented Programs Proc. Conf. Programming Language Design and Implementation (PLDI'03), 2003
4. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, vol. 15, pages 391–411, 1997
5. Artho, C., Havelund, K., Biere, A.: High-Level Data Races, VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems, April 2003
6. Walthert, S., Entwicklung eines Style Layers und Renderers für die XML-basierte GUI-Shell des Aos Systems, Diploma thesis, ETH Zürich, Switzerland 2001

LL(1) Conflict Resolution in a Recursive Descent Compiler Generator

Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck

Johannes Kepler University Linz, Institute of Practical Computer Science
Altenbergerstr. 69, 4040 Linz, Austria
{woess,loeberbauer,moessenboeck}@ssw.uni-linz.ac.at

Abstract. Recursive descent parsing is restricted to languages whose grammars are LL(1), i.e., which can be parsed top-down with a single lookahead symbol. Unfortunately, many languages such as Java, C++, or C# are not LL(1). Therefore recursive descent parsing cannot be used or the parser has to make its decisions based on semantic information or a multi-symbol lookahead.
In this paper we suggest a systematic technique for resolving LL(1) conflicts in recursive descent parsing and show how to integrate it into a compiler generator (Coco/R). The idea is to evaluate user-defined boolean expressions, in order to allow the parser to make its parsing decisions where a one symbol lookahead does not suffice.
Using our extended compiler generator we implemented a compiler front end for C# that can be used as a framework for implementing a variety of tools.

1 Introduction

Recursive descent parsing [16] is a popular top-down parsing technique that is simple, efficient, and convenient for integrating semantic processing. However, it requires the grammar of the parsed language to be LL(1), which means that the parser must always be able to select between alternatives with a single symbol lookahead. Unfortunately, many languages such as Java, C++ or C# are not LL(1) so that one either has to resort to bottom-up LALR(1) parsing [5, 1], which is more powerful but less convenient for semantic processing, or the parser has to resolve the LL(1) conflicts using semantic information or a multi-symbol lookahead.

In this paper we suggest a systematic technique for resolving LL(1) conflicts in recursive descent parsing and show how to integrate it into an existing compiler generator (Coco/R). Section 2 gives a brief overview of Coco/R. In Sect. 3 we explain the LL(1) property of grammars and show how to resolve certain LL(1) conflicts by grammar transformations. For those conflicts that cannot be eliminated, Sect. 4 introduces our conflict resolution technique, which is based on evaluation of user-defined boolean expressions. Section 5 describes a C# compiler front end built with our extended version of Coco/R, while Section 6 discusses related work. Finally, Sect. 7 summarizes the results and points out future plans.

2 The Compiler Generator Coco/R

Coco/R [8] is a compiler generator, which takes an attributed grammar and produces a scanner and a recursive descent parser from it. The user has to add classes (e.g. for symbol table handling, optimization and code generation), whose methods are called from the semantic actions of the attributed grammar.

Attributed grammars were introduced by Knuth [6] as a notation for describing the translation of languages. Originally they were declarative in nature. However, we use them as procedural descriptions. In this form, an attributed grammar consists of the following parts:

- A *context-free grammar* in EBNF as introduced by Wirth [15]. It describes the syntax of the language to be processed.
- *Attributes* that can be considered as parameters of the nonterminal symbols in the grammar. Input attributes provide information from the context of the nonterminal while output attributes deliver results computed during the processing of the nonterminal. In Coco/R attributes are enclosed in angle brackets (<...>).
- *Semantic actions*, which are statements in an imperative programming language (e.g. C#) that are executed during parsing. They compute attribute values and call methods, e.g. for symbol table handling or code generation. In Coco/R semantic actions are enclosed by (. and .).

Every production of an attributed grammar is processed from left to right. While the syntax is parsed the semantic actions are executed where they appear in the production. Here is an example of a production that processes variable declarations:

```
VarDeclaration             (. Structure type; string name; .)
= Type<out type>
  Ident<out name>          (. SymTab.Enter(name, type); .)
  { "," Ident<out name>    (. SymTab.Enter(name, type); .)
  } ";".
```

In order to increase the readability, the syntax parts are written on the left-hand side while semantic actions are written on the right-hand side. Curly braces ({...}) denote repetition, square brackets ([...]) denote optional parts, and vertical bars (|) separate alternatives. Coco/R translates the above production into the following recursive descent parsing routine where `sym` is the statically declared lookahead symbol:

```
static void VarDeclaration() {
  Structure type; string name;
  Type(out type);
  Ident(out name); SymTab.Enter(name, type);
  while (sym == Tokens.comma) {
     Scanner.Scan();
     Ident(out name); SymTab.Enter(name, type);
  }
  Expect(Tokens.semicolon);
}
```

Further information on how to use Coco as well as its source code and the executable can be obtained from [9, 14].

3 LL(1) Conflicts

A grammar is said to be LL(1) (i.e., parsable from **L**eft to right with **L**eft-canonical derivations and **1** symbol lookahead), if at any point, where the grammar allows a selection between two or more alternatives, the set of terminal start symbols of those alternatives are pairwise disjoint.

In EBNF grammars, there are the following three situations where LL(1) conflicts can occur (greek symbols denote arbitrary EBNF expressions such as `a[b]C`; *first*(α) denotes the set of terminal start symbols of the EBNF expression α; *follow*(A) denotes the set of terminal symbols that can follow the nonterminal A):

- **Explicit Alternatives**
  e.g. A = α | β | γ. check that *first*(α) ∩ *first*(β) = {} ∧ *first*(α) ∩ *first*(γ) = {} ∧ *first*(β) ∩ *first*(γ) = {}.
- **Options**
  e.g. A = [α] β. check that *first*(α) ∩ *first*(β) = {}
  e.g. A = [α]. check that *first*(α) ∩ *follow*(A) = {}
- **Iterations**
  e.g. A = {α} β. check that *first*(α) ∩ *first*(β) = {}
  e.g. A = {α}. check that *first*(α) ∩ *follow*(A) = {}

Resolving LL(1) Conflicts

Factorization. LL(1) conflicts can usually be resolved by factorization, i.e. by extracting the common parts of conflicting alternatives and moving them to the front. For example, the production

```
A = a b c | a b d.
```

can be transformed to

```
A = a b (c | d).
```

Left Recursion. Left recursion always represents an LL(1) conflict. In the production

```
A = A b | c.
```

both alternatives start with `c` (because *first*(`A`) = {`c`}). However, left recursion can always be transformed into an iteration, e.g. the previous production becomes

```
A = c {b}.
```

Hard Conflicts. Some LL(1) conflicts cannot be resolved by grammar transformations. Consider the following (simplified) production taken from the C# grammar:

```
IdentList = ident {"," ident} [","].
```

The conflict arises, because both the iteration and the option can start with a comma. There is no way to get rid of this conflict by transforming the grammar. The only way to resolve it, is to look ahead two symbols in order to see what follows after the comma. We will deal with this problem in Sect. 4.

Readability Issues. Some grammar transformations can degrade the readability of the grammar. Consider the following example (again taken from a simplified form of the C# grammar):

```
UsingClause = "using" [ident "="] Qualident ";".
Qualident   = ident {"." ident}.
```

The conflict is in `UsingClause` where both the option and `Qualident` start with `ident`. Although this conflict could be eliminated by transforming the production to

```
UsingClause = "using" ident ( {"." ident}
                            | "=" Qualident
                            ) ";".
```

the readability would clearly deteriorate. It is better to resolve this conflict as shown in Sect. 4.

Semantic issues. Finally, factorization is sometimes inhibited by the fact that the semantic processing of conflicting alternatives differs, e.g.:

```
A = ident (. x = 1; .) {"," ident (. x++; .) } ":"
  | ident (. Foo(); .) {"," ident (. Bar(); .) } ";".
```

The common parts of these two alternatives cannot be factored out, because each alternative has its own way to be processed semantically. Again this problem can be solved with the technique explained in Sect. 4.

4 Conflict Resolvers

This section shows how to resolve LL(1) conflicts by so-called *conflict resolvers*, which are in a similar form also used in other compiler generators (e.g. JavaCC, ANTLR, see Sect. 6 for a description of their approaches).

A conflict resolver is a boolean expression that is inserted into the grammar at the beginning of the first of two conflicting alternatives and decides by a multi-symbol lookahead or by semantic checks, whether this alternative matches the actual input. If the resolver yields true, the alternative is selected, otherwise the next alternative will be checked. A conflict resolver is given in the form

```
Resolver = "IF" "(" {ANY} ")" .
```

where `{ANY}` means an arbitrary piece of code that represents a boolean expression. In most cases this will be a method call that returns true or false.

Thus we can resolve the LL(1) conflict from Sect. 3 in the following way:

```
UsingClause = "using" [IF(IsAlias()) ident "="]
              Qualident ";".
```

`IsAlias` is a user-defined method that reads two symbols ahead. It returns true, if ident is followed by "=", otherwise false.

4.1 Multi-symbol Lookahead

The Coco/R generated parser remembers the last recognized terminal symbol as well as the current lookahead symbol in the following two global variables:

```
Token t;     // last recognized terminal symbol
Token la;    // lookahead symbol
```

In case one wants to look ahead more than just one symbol, the Coco/R generated scanner now offers the following two methods to do this:

- *StartPeek()* initializes peeking in the scanner by synchronizing it with the position after the current lookahead symbol.
- *Peek()* returns the next symbol as a *Token* object. The symbols returned by *Peek* are not removed from the input stream, so that the scanner will deliver them again when "normal" parsing resumes.

Using these methods we can implement *IsAlias* in the following way:

```
static bool IsAlias () {
  Scanner.StartPeek();
  Token x = Scanner.Peek();
  return la.kind == Tokens.ident && x.kind == Tokens.eql;
}
```

The last conflict mentioned in Sect. 3 can be resolved by the grammar rule

```
A = IF(FollowedByColon())
      ident (. x = 1; .) {"," ident (. x++; .) } ":"
    | ident (. Foo(); .) {"," ident (. Bar(); .) } ";".
```

and the following implementation of the function *FollowedByColon*:

```
static bool FollowedByColon () {
  Scanner.StartPeek();
  Token x = la;
  while (x.kind == Tokens.comma || x.kind == Tokens.ident)
    x = Scanner.Peek();
  return x.kind == Tokens.colon;
}
```

4.2 Conflict Resolution with Semantic Information

A conflict resolver can base its decision not only on the result of checking arbitrarily many lookahead symbols but any other kind of information as well. For example it could access a symbol table or other semantic information.

The following LL(1) conflict between assignments and declarations can be found in many programming languages:

```
Statement = Type IdentList ";"
          | ident "=" Expression ";"
          | … .
Type      = ident | … .
```

Both of the shown alternatives of the *Statement* rule begin with *ident*. This conflict can be resolved by checking whether ident denotes a type or a variable:

```
Statment = IF(IsType()) Type IdentList ";"
         | ident "=" Expression ";"
         | … .
```

IsType looks up *ident* in the symbol table and returns true, if it is a type name:

```
static bool IsType () {
  if (la.kind == Tokens.ident) {
    object obj = SymTab.find(la.val);
    return obj != null && obj.kind == Type;
  } else return false;
}
```

4.3 Translation of Conflict Resolvers into the Parser

Coco/R treats conflict resolvers similarly to semantic actions and simply copies them into the generated parser at the same position where they appear in the grammar. The production for the *UsingClause* from above is translated into the following parser method:

```
static void UsingClause () {
  Expect(Tokens.using);
  if (IsAlias()) {
    Expect(Tokens.ident); Expect(Tokens.eql);
  }
  Qualident();
  Expect(Tokens.semicolon);
}
```

4.4 A Different Approach: Resolver Symbols

In a previous version of Coco/R, we have examined a different approach to resolve LL(1) conflicts: artificial tokens – so-called *resolver symbols* – were inserted into the input stream on-the-fly (at parse time) in order to guide the parser along the right way. The symbols had to be placed properly in the grammar rules. They were defined in a separate section of the grammar (keyword: `RESOLVERS`) and combined with a resolution routine that determined, whether the resolver symbol was needed. If so, the parser inserted it into the input stream. Coco/R put the invocations of the resolution routines automatically at the right places in the generated parser.

Comparing the two approaches, we find that the current conflict resolver version (`IF(…)`) requires less implementation changes compared to the original Coco/R (without conflict resolution), is less complex and appears to be easier to understand from the users perspective than the insertion of artificial tokens at parse time.

5 A Compiler Front End for C#

In a project supported by Microsoft under the Rotor initiative [11] we used Coco/R and its conflict resolution technique for building a compiler framework for C# that can be used to build compilers, source code analyzers and other tools. Our framework consists of an attributed C# grammar to which the user can attach semantic actions as required for the tool that he is going to implement. From this description Coco/R generates a scanner and a parser that also contains the attached semantic actions.

We started out by downloading the C# grammar from the ECMA standardization page [12]. Since this grammar was designed for being processed by a bottom-up parser, it is full of left recursions and other LL(1) conflicts. Some of these conflicts could be resolved by grammar transformations as discussed in Section 3, the others were eliminated using conflict resolvers. The resulting grammar is available at [17 or 3].

One of the first applications that have been developed with our framework is a white box testing tool [7] that instruments a C# program with counters in order to get path coverage information. Another application analyzes C# source code to provide helpful hints as to where performance optimizations could be applied or coding style rules have been violated [13]. Other applications that could be implemented with our framework are for example:

- Profilers that measure the execution frequencies or execution times of C# programs.
- Source code analyzers that compute complexity measures or data flow information from C# programs.
- Pretty printers that transform a C# program according to certain style guidelines.
- Proper compilers that translate C# programs to IL, machine code, Java bytecodes, or to XML schemata.

6 Related Work

Recursive descent parsers are usually written by hand, because this is straightforward to do. In manually written parsers it is common to resolve LL(1) conflicts by semantic information or by multi-symbol lookahead. However, these are only ad-hoc solutions and do not follow a systematic pattern as we described it in Section 4.

Compiler generators, on the other hand, usually generate bottom-up parsers that use the more powerful LALR(1) technique instead of LL(1) parsing. In bottom-up parsers, however, it is more difficult to integrate semantic processing, because semantic actions can only be performed at the end of productions.

Recently, there seems to be a growing interest in compiler generators that produce recursive descent parsers. In the following subsections we will describe two of these generators and their LL(1) conflict resolution strategies.

6.1 JavaCC

JavaCC [4] is a compiler generator that was jointly developed by Metamata and Sun Microsystems. It produces recursive descent parsers and offers various strategies for resolving LL(1) conflicts.

First, one can instruct JavaCC to produce an LL(k) parser with $k > 1$. Such a parser maintains k lookahead symbols and uses them to resolve conflicts. Many languages, however, not only fail to be LL(1), but they are not even LL(k) for an arbitrary, but fixed k. Therefore JavaCC offers local conflict resolution techniques. A local conflict resolver can be placed in front of alternatives in the same way as in Coco/R. The following syntax is available (greek letters denote arbitrary EBNF expressions):

- (LOOKAHEAD(k) $\alpha \mid \beta$) tells the parser that it should look ahead k symbols in the attempt to recognize an α.
- (LOOKAHEAD(γ) $\alpha \mid \beta$) allows the user to specify an EBNF expression γ. If the next input symbols match this expression, then α is selected.
- (LOOKAHEAD(*boolExpr*) $\alpha \mid \beta$) allows the user to specify a boolean expression (which can also be the call to a function that returns true or false). If the expression evaluates to true, then α is selected. This is the only strategy currently supported by Coco/R.

So JavaCC offers more ways of how to resolve LL(1) conflicts than does Coco/R. Other differences are:

- JavaCC produces parsers in Java while Coco/R produces parsers in C#. Furthermore, the new conflict resolution feature will be incorporated in the Java version of Coco/R and a complete port to VB.NET is also on the agenda.
- Coco/R uses Wirth's EBNF [15], while JavaCC uses an EBNF notation that is similar to regular expressions.

6.2 ANTLR

ANother Tool for Language Recognition (ANTLR) [2, 10] is part of the Purdue Compiler Construction Tool Set (PCCTS), which includes also SORCERER (a tree parser generator) and DLG (a scanner generator). ANTLR generates recursive descent parsers from so-called "pred-LL(k)" grammars, where "pred" indicates that syntactic or semantic predicates are used to direct the parsing process. The current version of ANTLR can produce parsers in Java, C++, and C#.

Syntactic predicates allow the user to specify a syntax expression in front of an alternative. If this expression is recognized, the alternative is selected (this is similar to JavaCC's syntactic LOOKAHEAD feature (the second in the above list)):

```
stat: (list "=" )=> list "=" list
    |               list ;
```

Semantic predicates are used in two ways: Firstly, for checking context conditions, throwing an exception, if the conditions are not met:

```
decl: "var" ID ":" t:ID
      { isTypeName(t.getText()) }? ;
```

Secondly, they are used for selecting between alternatives:

```
stat: { isTypeName(LT(1)) }?
      ID ID ";"             // declaration (Type ident;)
    | ID "=" expr ";" ;     // assignment
```

Both cases have the same syntax. They are only distinguished by the position of the predicate: disambiguating predicates must appear at the beginning of an alternative.

ANTLR's approach is similar to that of JavaCC and Coco/R. All evaluate boolean expressions in order to guide the parsing process, i.e., the user is given control to define the conditions under which an alternative shall be selected.

7 Summary

In this paper we showed how to resolve LL(1) conflicts systematically by using boolean expressions – so-called *conflict resolvers* – that are evaluated at parse time in order to guide the parser along the right path. In this way the parser can exploit either semantic information or on-demand multi-symbol lookahead in its decision-making process. We also described the integration of our technique into the compiler generator Coco/R.

The work described in this paper is ongoing research. In the future we plan to add more resolution strategies to Coco/R in a similar way as it is done for JavaCC. For example, we plan to implement a feature that does an LL(k) lookahead without requiring the user to specify a resolver routine. We hope that it will even be possible to let Coco/R decide, at which points a k-symbol lookahead is necessary and to do it automatically without any intervention of the user. Finally, since the original version of Coco/R is available for a variety of languages such as C#, Java, C++ and others, we plan to port our conflict resolution technique from the C# version of Coco/R also to the other versions and offer a new VB.NET version. The current state of our project as well as downloads of the source code and the executables of Coco/R are available at [17, 3].

Acknowledgements. This work was supported by Microsoft under the Rotor initiative [11].

References

1. Aho A.V., Ullman J.D.: Principles of Compiler Design, Addison-Wesley, 1977
2. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org
3. Coco Tools Project at the Rotor Community Site, http://cocotools.sscli.net
4. JavaCC™ (Java Compiler Compiler), The Java Parser Generator. http://www.experimentalstuff.com/Technologies/JavaCC
5. Knuth, D.E.: On the Translation of Languages from Left to Right. Information and Control, vol. 6, pp. 607–639, 1965
6. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory, vol 2, pp.127–145, 1968
7. Löberbauer, M.: Ein Werkzeug für den White-Box-Test. Diploma thesis, Institute of Practical Computer Science, University of Linz, Austria, February 2003
8. Mössenböck, H.: A Generator for Production Quality Compilers. 3rd intl. workshop on compiler compilers (CC'90), Schwerin, Lecture Notes in Computer Science 477, Springer-Verlag, 1990, pp. 42–55.
9. Mössenböck, H.: Coco/R for various languages – Online Documentation. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
10. Parr, T.: Language Translation Using PCCTS and C++. Automata Publishing Company, 1993.
11. Rotor: The Microsoft Rotor Project, http://research.microsoft.com/collaboration/university/europe/rotor/
12. Standard ECMA-334, C# Language Specification, December 2001, http://www.ecma-international.org/publications/standards/ecma-334.htm
13. Steineder, K.H.: Optimization Hint Tool. Diploma thesis, Institute for Practical Computer Science, University of Linz, Austria, August 2003.
14. Terry, P.: Compilers and Compiler Generators – An Introduction Using C++. International Thomson Computer Press, 1997.
15. Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, November 1977
16. Wirth, N.: Compiler Construction. Addison-Wesley, 1996.
17. Wöß, A., Löberbauer, M.: SSCLI-Project: Compiler Generation Tools for C#. http://dotnet.jku.at/Projects/Rotor/

Graph Coloring vs. Optimal Register Allocation for Optimizing Compilers*

Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz

Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
{uli,andi,scholz}@complang.tuwien.ac.at

Abstract. Optimizing compilers play an important role for the efficient execution of programs written in high level programming languages. Current microprocessors impose the problem that the gap between processor cycle time and memory latency increases. In order to fully exploit the potential of processors, nearly optimal register allocation is of paramount importance. In the predominance of the x86 architecture and in the increased usage of high-level programming languages for embedded systems peculiarities and irregularities in their register sets have to be handled. These irregularities makes the task of register allocation for optimizing compilers more difficult than for regular architectures and register files. In this article we show how optimistic graph coloring register allocation can be extended to handle these irregularities. Additionally we present an exponential algorithm which in most cases can compute an optimal solution for register allocation and copy elimination. These algorithms are evaluated on a virtual processor architecture modeling two and three operand architectures with different register file sizes. The evaluation demonstrates that the heuristic graph coloring register allocator comes close to the optimal solution for large register files, but performs badly on small register files. For small register files the optimal algorithm is fast enough to replace a heuristic algorithm.

1 Introduction and Motivation

Nowadays high level programming languages dominate the software development for all kinds of applications on a broad range of processor architectures. For the efficient execution of programs optimizing compilers are required whereby one of the most important components in the compiler is register allocation. Register allocation maps the program variables to CPU registers. The objective of an allocator is to assign all variables to CPU registers. However, often the number of registers is not sufficient and some variables have to be stored in slow memory (also known as *spilling*).

Traditionally, register allocation is solved by employing graph coloring that is a well-known NP-complete problem. Heuristics are used to find good register allocation in nearly linear time [4,3].

* This research is partially supported by the Christian Doppler Forschungsgesellschaft as part of the Project "Compilation Techniques for Embedded Processors"

For regular architectures with large register sets graph coloring register allocation is well explored. In general the heuristic algorithms produce very good results and register allocations come close to the optimal solution [6]. With small and irregular register sets register allocation by graph coloring has to be adapted.

A prominent example of an irregular architecture with a small number of registers is the x86 which has overlapping registers and special register usage requirements. Another example is the Motorola 68k architecture which has separate register files for data and addresses. Both architectures have two-operand instructions whereas more modern RISC architectures provide three-operands in the instruction set. Intel's IA-64 processors feature VLIW (Very-Large Instruction Word), predicated execution, and rotating register files. Furthermore, embedded processors and digital signal processors (DSP) have non-orthogonal instruction sets and impose additional constraints for register allocation.

2 Processor Models

2.1 General Description

Our processor is a 5 way variable length VLIW load/store architecture. It supports some commonly met extensions for the DSP domain, like multiply accumulate instructions, various addressing modes for loads and stores, fixed point arithmetic, predicated execution, SIMD instructions, etc. The processor's register file consists of two distinct register banks, one bank for data registers, the other one for address registers. Each data register is 40 bit wide, but can also be used as a 32 bit register, or as two registers of 16 bit width ("shared registers", "overlapping registers", "register pairs"). Address registers are 16 bit wide.

The register file layout described above causes several idiosyncrasies in the instruction set. Not all of the instructions may use both registers types. E.g., a multiplication may not use address registers, or load and stores can get their addresses only from address registers (hence the name). Only simple integer arithmetic is possible with address registers. Some instructions require their operands to be assigned to adjacent register pairs (SIMD).

2.2 Model Parameters

The first parameter is the number of registers per bank. We modeled 4, 8, and 16 registers per bank, respectively. This covers many of todays popular processors in the embedded domain. See Fig. 1 for the register model.

The second parameter is the number of operands per instruction. The quasi–standard mode is three operand mode, meaning that instructions can address up to 3 different registers (two source registers and one destination register in case of a binary operation). Two operand mode only allows to address two registers per instruction. Binary operations therefore require one of the source operands being the same as the destination operand. This can easily be achieved by inserting additional register move instructions prior to binary operations.

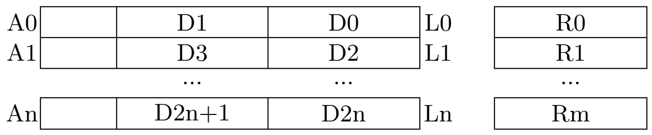


Fig. 1. Register file for model 8R/xA

3 Extension to Graph Coloring

Chaitin [4] was the first who used a graph coloring approach for register allocation. Many of the ideas and phases of graph coloring register allocation introduced by him are used in current register allocators. It is assumed that an unlimited number of symbolic registers is available to store local variables and large constants. During register allocation these symbolic registers are mapped to a finite number of machine registers. If there are not sufficient machine registers some of the symbolic registers are stored in the memory (spilled registers). Liveness analysis determines the live ranges of symbolic registers. Symbolic registers which are live at the same time cannot be assigned to the same machine register. These assignment constraints are stored in an interference graph. The nodes of this graph represent the symbolic registers. An edge connects two nodes, if these two symbolic registers are live at the same time. After liveness has been computed live ranges are combined when they are connected by a copy instruction (coalescing). Graph coloring register allocation colors the interference graph with machine registers spilling some symbolic registers to memory. The register allocator has to respect constraints like certain register requirements or pairing of registers.

3.1 Irregular Processors

Graph coloring register allocation becomes complicated when processors have nonorthogonal register sets or restrictions. For example paired registers form registers of twice the size. There are instructions which can only use a certain register or a subset of registers. Two operand instructions require that the destination register of an instruction is the same as one of the source registers.

Briggs [2] solved the problem of paired registers using a multigraph. A paired register has two edges to another register and a single register has a single edge to another single register. Such a scheme models paired registers without changing the other parts of the algorithm or the heuristics. The colorability of a graph can be determined by the degree of a node as before. Smith and Holloway suggested using a weighted interference graph and we followed their approach [12].

When using a weighted interference graph the nodes are augmented with a weight which corresponds to the occupied register resources. The edges represent the program constraints whereas the weights represent the architectural constraints. In a weighted graph minor changes to the algorithm are necessary. The colorability of a node cannot be determined by the degree anymore. Since we are using optimistic coloring where spill decisions are determined in the select phase exact computation of the colorability is not necessary. Because we are already using inaccurate heuristics, we should use exact computations when possible.

We are weighting short registers with the weight 1 and long registers or accumulators with weight 2. The colorability equation has to take the weight of the node and of its neighbors into account. Equation 1 gives the details of the computation.

$$d_n = \sum_{j \, \epsilon \, adj(n)} \lceil \frac{w_j}{w_n} \rceil * w_n \tag{1}$$

d_n is the virtual degree of the weighted node n. If d_n is smaller than N, it is guaranteed that the node can be colored (N is the number of CPU registers multiplied with the applied weight for n's register type).

The assignment of certain kinds of values maybe restricted to a predefined set of registers. This is achieved by precoloring these registers to the required machine registers [5]. To ensure that coloring a graph with precolored registers is possible we add move instructions between the symbolic register and the machine register. If possible these move instructions are eliminated during coalescing. A typical example are argument registers. At the entrance of a function the incoming argument registers are copied to symbolic registers and before a function invocation symbolic registers are copied to the outgoing arguments registers. Similarly before the end of a function the symbolic register is copied to the return value register. Machine instructions like a multiply-accumulate with a fixed accumulator register also get this operand precolored and move instructions are added to ensure colorability.

Some values have to stay in a subset of the machine registers. Examples are symbolic registers which are live across a function call or symbolic registers which are operands of machine instructions which only accept a subset of the register set. Additionally to the symbolic registers the interference graph contains all machine registers. If only a subset of the registers is allowed, interference edges are added to the complement set of the allowed machine registers. Symbolic registers which are live across a function call are in conflict with all machine registers except callee saved registers. As a further optimization these registers are saved in caller saved registers instead of memory when caller saved registers are available and the cost of spilling to memory is higher.

3.2 Coalescing

If source and destination of a move instruction do not interfere, the move can be eliminated and its operands can be combined into one live range. The interferences of the new live range are the union of those of the live ranges being combined. Coalescing of live ranges can prevent coalescing of other live ranges. Therefore coalescing order is important. Move instructions with higher execution frequencies should be eliminated first.

Aggressive coalescing combines any of non interfering copy related live ranges. This reckless strategy may turn a k–colorable graph into a not k–colorable graph and thus is unsafe. Briggs suggests in [1] only to combine live ranges that result in a live range that has fewer than k neighbors of degree k or higher. George and Appel suggest in [7] to combine a and b only if for every neighbor t of a, either t already interferes with b or the degree of t is less than k. Additionally, they propose interleaving coalescing and graph simplification. Both strategies are safe, but the iterative approach of George and Appel achieves better coloring results.

If one of the copy related live ranges is precolored, non interference is not a sufficient prerequisite to do the combination. Suppose a live range a that is precolored to R and a copy related live range b. a and b can only be coalesced if neither of b's neighbors is precolored to R. Further constraints on coalescing arise from the irregular architecture. It is only possible to combine live ranges of equal types. Interbank moves (between data and address bank) cannot be coalesced at all. Combining live ranges of shorts can be problematic, too. Any of them can be part of a paired register. They can only be combined, if there are no such constraints, or if they have equal constraints. If only one is constrained, the constraint must be propagated to the resulting live range.

Two operand mode requires to copy one source operand of a binary operation to the destination operand. In case of commutative operations, choosing a source operand that does not interfere with the destination should be favored. The inserted move will later be eliminated.

3.3 Complete Algorithm

The complete algorithm executes the single phases linearly and does an iteration if spilling is necessary. The single phases are liveness analysis with interference graph construction, coalescing, interference graph construction and coloring. When adding spill instructions care has to be taken to avoid endless looping because of spilling live ranges which occur due to loading spilled registers.

4 Optimal Register Allocation

Optimal register allocation delivers the most accurate register allocation. By its nature register allocation is a very hard problem to solve and an optimal solution requires exponential run-time for its computation. Therefore, production compilers sacrifice optimality in order to obtain a fast register allocation. Heuristics are applied which give sub-optimal answers.

However, an optimal register allocation scheme is relevant for assessing heuristics how good they work in practice. When traditional graph-coloring heuristics [3], which have an excellent performance for RISC architectures with a reasonable number of registers, are extended for architectural peculiarities, the quality of the register allocation might suffer and the comparison with the optimal solution is essential.

Instead of using an *Integer Linear Programming(ILP)* [8,9,6] approach for obtaining an optimal solution, we employed a new algorithm [11] that is based on *Partitioned Boolean Quadratic Programming(PBQP).* The PBQP approach is a unified approach for register allocation that can model a wide range of peculiarities and supersedes traditional extended graph-coloring approaches [3, 12]. In addition coalescing is an integral part of the register assignment which is necessary for achieving good allocations.

The PBQP approach uses cost functions for register assignment decisions. The cost functions have to fulfill two tasks: (1) cost functions that express mathematically the model of cost of the architecture, and (2) cost functions that describe interference constraints, coalescing benefits, and constraints which stem from the CPU architecture. Basically, we have two classes of cost functions. One class of cost functions model the costs and constraints involved for one symbolic register, and the second class of cost functions model the costs and constraints of two dependent symbolic registers. For our processor we only need a sub set of the cost functions that were introduced in [11] since its architecture is fairly orthogonal.

In Table 1 the cost functions for our processor architecture are listed. Spilling is modeled by a cost function for one symbolic register. The parameter c gives the costs for spilling and parameter a determines the allocation of symbolic register $\mathtt{s}$. A symbolic register is either spilled (i.e. a is equal to sp) or a register is assigned to it (i.e. $a \in \{R_0, R_1, \dots\}$). Depending on this decision different costs are involved. The architecture features four register classes which can be modeled by $c_{\mathtt{s}}(a)$. Registers which are disabled for a symbolic register have ∞ costs and therefore are excluded for register assignments. An interference of two symbolic registers $\mathtt{s}_1$ and $\mathtt{s}_2$ is given by $i_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2)$. Either both allocations have different register assignments ($a_1 \neq a_2$) or one of the registers is spilled ($a_1 = sp$). Again, infinite costs will be raised if for both symbolic register the same CPU register is allocated. The shared register interference constraint is given in equation for $d_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2)$. This constraint is necessary since two short registers share the same memory of one long register in the architecture. Coalescing costs of two symbolic registers are given by $p^{(b)}_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2)$. If both register assignments of $\mathtt{s}_1$ and $\mathtt{s}_2$ are identical we obtain a coalescing benefit expressed as a negative number $-b$.

The cost functions are used for constructing cost matrices and cost vectors for the PBQP problem. The NP-hard PBQP problem is given as follows:

$$\min f = \left[\sum_{1 \le i < j \le n} \boldsymbol{x}_i \cdot C_{ij} \cdot {\boldsymbol{x}_j}^T \right] + \left[\sum_{1 \le i \le n} \boldsymbol{c}_i \cdot {\boldsymbol{x}_i}^T \right]$$

$$\text{subject to: } \forall i \in 1 \dots n : \boldsymbol{x}_i \cdot \mathbf{1}^T = 1$$

The cost matrices C_{ij} are determined by the cost functions $F_{\mathtt{s}_i,\mathtt{s}_j}$ of symbolic registers $\mathtt{s}_i$ and $\mathtt{s}_j$ as follows:

$$\forall a_k, a_l \in A : C_{ij}(k,l) = \sum_{f_{\mathtt{s}_i\,\mathtt{s}_j} \in F_{\mathtt{s}_i\,\mathtt{s}_j}} f_{\mathtt{s}_i\,\mathtt{s}_j}(a_k, a_l)$$

Table 1. Cost functions for our processor

Spilling:

$$s^{(c)}_{\mathtt{s}}(a) = \begin{cases} c, & \text{if } a = sp \\ 0, & \text{otherwise} \end{cases}$$

Class Constraint:

$$c_{\mathtt{s}}(a) = \begin{cases} 0, & \text{if } a \in \mathit{class}(\mathtt{s}) \cup \{sp\}, \\ \infty, & \text{otherwise} \end{cases}$$

Interference:

$$i_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \neq a_2 \vee a_1 = sp \\ \infty, & \text{otherwise} \end{cases}$$

Shared Register(Interference)

$$d_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) = \begin{cases} \infty, & \text{if } a_1 \in \mathit{shared}(a_2) \\ 0, & \text{otherwise} \end{cases}$$

Coalescing:

$$p^{(b)}_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) = \begin{cases} -b, & \text{if } a_1 = a_2 \wedge a_1 \neq sp \\ 0, & \text{otherwise} \end{cases}$$

The cost vectors $\boldsymbol{c}_i$ are determined by the cost function $f_{\mathtt{s}_i}$ of symbolic register $\mathtt{s}_i$.

$$\forall a_k \in A : \boldsymbol{c}_i(k) = \sum_{f_{\mathtt{s}_i} \in F_{\mathtt{s}_i}} f_{\mathtt{s}_i}(a_k)$$

The PBQP problem can be solved by dynamic programming as proposed in [11]. In each step of the algorithm a vector $\boldsymbol{x}_i$ is eliminated until the objective function f becomes trivial, i.e. the first part of the sum $\sum_{1 \leq i < j \leq n} \boldsymbol{x}_i \cdot C_{ij} \cdot \boldsymbol{x}_j{}^T$ vanishes. Then, the solution of remaining vectors in the objective function is determined. Reduced vectors can be computed by reconstructing the original objective function. Unfortunately, not all reductions can be applied in polynomial time. Therefore, a recursively enumeration is necessary for obtaining the optimal solution. Basically, we have three reduction: reduction RI for nodes of only one cost matrix, reduction RII for nodes of two cost matrices, and reduction RN for nodes with arbitrary number of cost matrices. Reductions RI and RII can be solved in polynomial time — reduction RN needs exponential time.

The RN reduction for the optimal solution is given in Fig. 2. The first loop enumerates all possible solutions of vector x. For a given solution the costs are determined. If it is smaller than the current minimum the solutions of the

```
procedure ReduceN($x$)
begin
        $min := \infty$;
    for i:=1 to $|\boldsymbol{c}_x|$ do
        $h := \boldsymbol{c}_x(i)$;
        for all $y \in adj(x)$ do
            $\boldsymbol{c}_y := \boldsymbol{c}_y + C_x y(i,:)$;
        end
        remove $x$;
        for all $scc \in G$ do
            solve scc;
            $h := h$+cost(scc);
        endfor
        if $h < min$ then
            save solutions
        endif
        reconstruct node $x$
    endfor
    restore min. solution
end
```

Fig. 2. RN reduction

remaining vectors are saved. The reduction of the vector can split the PBQP graph in several independent sub-graphs (scc). The performance of the algorithm can be substantially improved by solving the independent sub-graphs on their own. For reducing the number of RN nodes it is a good heuristic to select the vector with the highest number of cost matrices.

5 Results

The intention of the experiments is to compare the implemented register allocation methods on a broad range of architectural models on a wide variety of programs. The complete evaluation is contained in a longer version of this article and available at `www.complang.tuwien.ac.at/papers/HiKrSch2003.ps`. We measured the obtained results both in terms of spilling costs and coalescing benefits, as well as solve times for coloring the interference graph.

Our experiments covered a total of 210 functions from typical DSP applications.

Since the optimal method has a worst case of $O(k^n)$ (with k ... number of registers , n ... number of nodes), we implemented a timeout. Each function is given 30 minutes to be solved. If no solution is found, the record is deleted from all the result sets, and only the remaining subsets of the qualitative records can be compared. Figure 3 shows the number of solved functions per model. The minimum is at model *16R/2O*, where 148 functions were solved (i.e. all

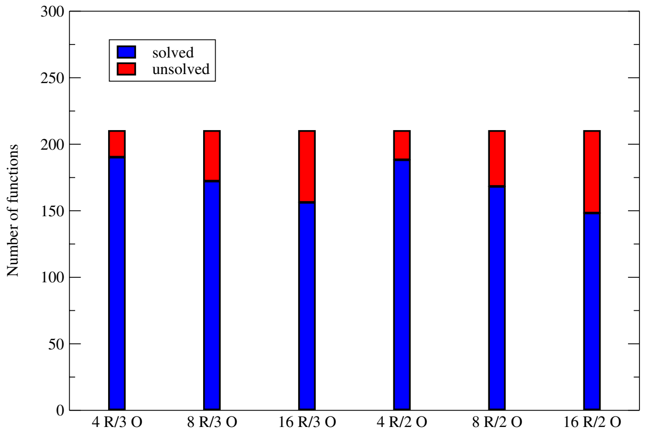


Fig. 3. Functions with optimal solution

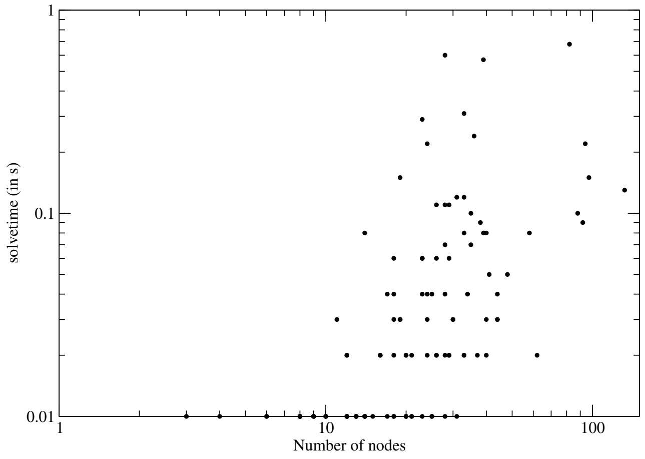


Fig. 4. Solve times for *4R/3O*

qualitative data refer to this subset of functions). The solve times for model *4R/3O* are presented in Fig. 4.

Key points of interest are spilling costs and coalescing benefits. We evaluated spilling costs and also counted the number of spilled live ranges. Optimal register allocation performs better in both terms. The gainings over graph coloring are best when the number of registers is small. The optimal allocator also achieves better coalescing benefits than graph coloring with aggressive coalescing. Be aware that the baseline of the coalescing benefits is not zero. It is known that aggressive coalescing may overly constrain live ranges, so that these cannot be colored and thus do not contribute to the coalescing benefit. See Figs. 5 and 6 for an overview.

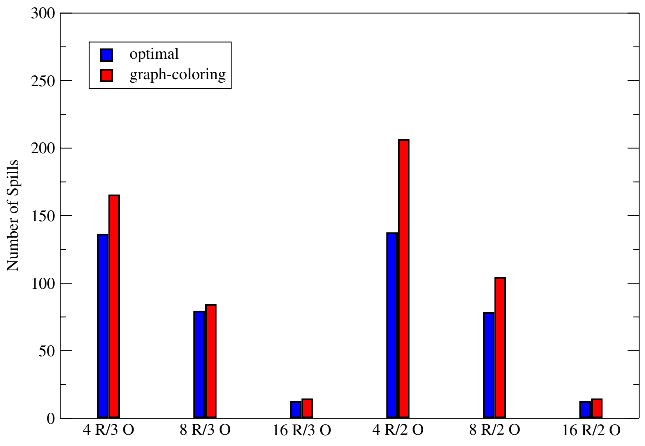


Fig. 5. Number of spills

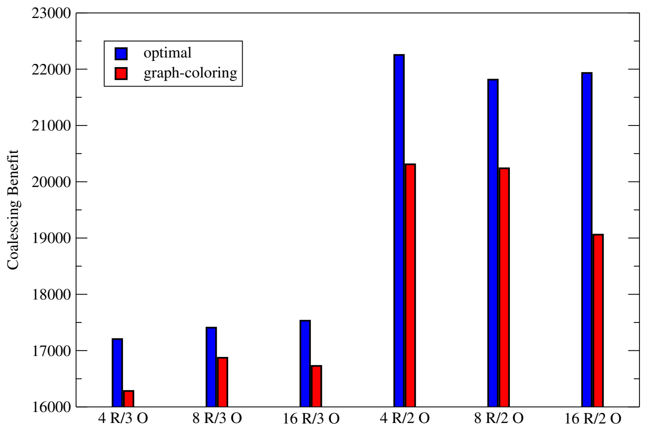


Fig. 6. Coalescing benefits

6 Related Work

Register allocation based on integer linear programming (ILP) was introduced by Goodwin and Wilken [8,6]. The approach maps the register allocation problem to an integer linear program which is solved by an NP-complete ILP-solver. The work was extended by Kong and Wilken [9] for irregular architectures. As an example they choose the IA-32 architecture and added additional features such as address mode selection. The approach can handle irregularities very nicely. However, the underlying algorithms have exponential solve time and the solvers are not able to solve bigger functions in reasonable time – they simple cut-off the solver.

Copy propagation is an important task for register allocators, which is achieved by assigning source and destination of a copy instruction the same registers. In the past several approaches as Chaitin's *aggressive coalescing*[4], Briggs' conservative coalescing [3], George and Appel's iterated coalescing[7], and Park and Moon's optimistic coalescing [10] were introduced. All have in common that they coalesce nodes of the interference graph in a separate pass. The node selection for coalescing and the strategy of uncolored and coalesced nodes differs depending on the approach.

7 Conclusions

In this work, we presented an extensive experimental evaluation of two different register allocation methods for irregular processor architectures. Our experiments show, that the graph coloring based algorithm causes more spilling costs than the algorithm with an optimal solution. Smaller register numbers result in an increase of this penalty. Further, aggressive coalescing does not result in higher coalescing benefits. It overly constrains some of the concerned live ranges and therefore forces additional spills.

Comparison purely by solve times makes graph coloring the winner. Most of the functions are colored in less time than we were able to measure, whereas PBQP is not even able to solve all the functions within a timeout of 30 minutes. This computation effort does not pay for architectures with a large register file, where graph coloring achieves near optimal results.

For architectures with a small register file, computation of the optimal solution runs acceptably fast, and the gainings over graph coloring are significant. In this case, it is practicable and feasible to chose the optimal method for most of the typical applications. A longer version of this article is available at `www.complang.tuwien.ac.at/papers/HiKrSch2003.ps`.

Acknowledgement. We like to thank the reviewers for their helpful suggestions.

References

1. K.D.C.P. Briggs, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989.
2. P. Briggs, K. D. Cooper, and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems, (LOPLAS)*, 1(1):3–13, Mar. 1992.
3. P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
4. G.J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982.
5. F.C. Chow and J.L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

6. C. Fu and K.D. Wilken. A faster optimal register allocator. In *Proceedings of the 35st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-02)*, pages 245–256, Istanbul, Nov. 18–22 2002. IEEE Computer Society.
7. L. George and A.W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
8. D.W. Goodwin and K.D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software & Practice and Experience*, 26(8):929–965, Aug. 1996.
9. T. Kong and K.D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, Nov. 30–Dec. 2 1998. IEEE Computer Society.
10. J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 196–204, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.
11. B. Scholz and E. Eckstein. Register allocation for irregular register architectures. In *Proceedings of the International Conference of Languages, Compilers and Tools for Embedded Systems (LCTES'02) and SCOPES'02*, Berlin, June 2002. ACM.
12. M. D. Smith and G. Holloway. Graph-coloring register allocation for irregular architectures. Technical report, Harvard University, 2000.

A Source-to-Source Architecture for User-Defined Optimizations

Markus Schordan and Dan Quinlan

Lawrence Livermore National Laboratory, CA 94551, USA
schordan1@llnl.gov, dquinlan@llnl.gov

Abstract. We present an architecture for the specification of source-to-source transformations. New source code can be specified as source-fragments. The translation of source-fragments to the intermediate representation is accomplished by invoking the frontend. For any inserted fragment we can guarantee that it is typed correctly. If no error is reported on inserted fragments, the whole program can always be compiled without errors. Based on a given abstract attribute grammar the user can specify transformations as semantic actions and can combine the computation of attributes with restructure operations on the intermediate representation.

1 Introduction

The development of special purpose (domain-specific) libraries to encapsulate the complexity of software is a significant step toward the simplification of software. But the abstractions presented by such libraries are user-defined and not optimized by the vendor's language compiler. The economics and maturation of new language and compiler designs make it particularly difficult for highly specialized languages to appear and be accepted by developers of large scale applications. Unfortunately, the generally poor level of optimization of user-defined abstractions within applications thus negates their effective widespread use in fields where high performance is a necessity.

Though significant aspects of our approach are language independent, our research work has targeted the optimization of C++ applications. The framework developed to support this research, ROSE, [1], allows us to express optimizations based on an abstract C++ grammar, eliminating the syntactical idiosynchrases of C++ in the specification of a transformation. Because we target library developers generally, our approach avoids the requirement that users learn a new special purpose language to express transformations. The semantic actions which specify a transformation are implemented in C++.

Within previous research we have demonstrated the use of ROSE [1,2] and that the performance penalty of user-defined abstractions can be overcome by source-to-source transformations. We presented how a speedup of up to four can be achieved for user-defined abstractions as they are used in practice. The use of the semantics of the user-defined abstractions has been an essential part of this success. In this paper we demonstrate the use of the abstract grammar in combination with source strings and restructuring of the intermediate representation. As example we discuss the core of an OpenMP parallelization. The high-level semantics of the user-defined type utilized in the example is the thread-safety of its methods.

In Sect. 2 we describe the architecture and how we can translate incomplete source-fragments to corresponding fragments of the intermediate representation. In Sect. 3 we discuss how program transformations are specified by the use of the abstract grammar and source-strings. In the final sections we discuss related research and our conclusions.

2 Source-to-Source Architecture

In a usual source-to-source translation the frontend is invoked once. Transformations are either syntax directed or defined as explicit operations on an intermediate representation (IR). Eventually the backend is called to generate the final source program. In our architecture, see Fig. 1, the frontend and backend are components that can be invoked at any point in an operation on the IR to obtain program fragments.

The capability of translating source-fragments to IR-fragments and back is essential to allow a compact specification of transformations as demonstrated in the example in Sect. 3. This allows the definition of a transformation by combining sequential strings although our intermediate representation has a tree structure. Although strings are used, by invoking the frontend each fragment is type-checked before it is inserted in the IR. This ensures that in each step of a transformation, when a part of the intermediate representation is replaced by a new fragment, the program fragment is checked for syntactical and semantical correctness.

The combination of different source-fragments is specified in semantic actions associated with rules of an abstract grammar. The computed attribute values can be of arbitrary type, including source-fragments. Because the computed attributes can also be source-fragments it is necessary to translate them to IR-fragments to insert them into the IR. Note, we do not re-parse source-strings, a source-string is only parsed once by the frontend. But the frontend can be invoked to translate computed source-strings, ensuring that all semantic checks are performed on the inserted IR-fragments as well. Note that our approach does not require any modifications to an existing frontend.

We use the EDG-frontend [3] for parsing C++ programs. This frontend performs template instantiation and a full type evaluation. In our abstract grammar all type information is made available to the user as annotations of nodes in the abstract syntax tree (AST) which can be accessed in semantic actions of the abstract attribute grammar. The availability of semantic compile-time information is an essential aspect of our architecture. In the following sections we describe in detail how source-fragments can be translated to IR-fragments by utilizing an existing frontend and how all semantic information can be updated in the IR.

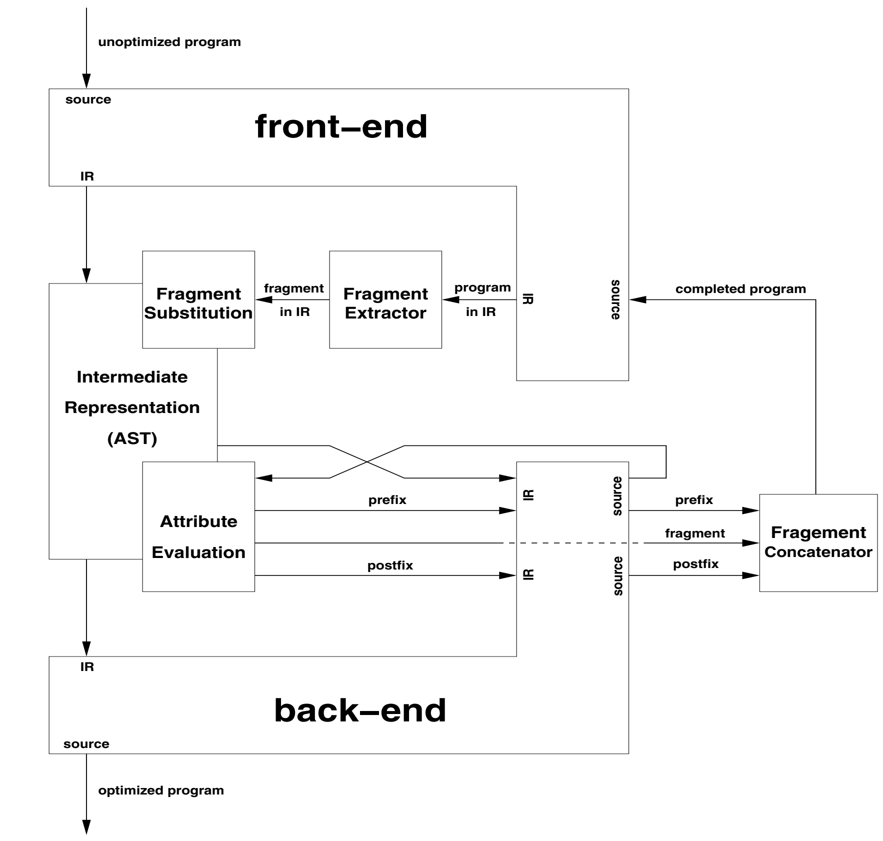


Fig. 1. Source-to-source architecture with frontend/backend invocation

2.1 Fragment Concatenator and Extractor

In general, a source-fragment cannot be parsed by the frontend because it is an incomplete program. Therefore it needs to be extended by a source-prefix and a source-postfix to a complete program such that it can be parsed by the frontend. This computation of the prefix and postfix is automated. The user only specifies the fragment and the target location of the corresponding IR-fragment. In our IR, the target location, L_{abs}, is a node in the AST. The prefix and postfix are automatically generated. The source-prefix consists of all declarations and opening braces of scopes before the target location, the source-postfix consists of all closing braces of scopes after the target position.

The frontend returns a program in IR. From this the corresponding IR-fragment needs to be extracted. A source string shall be denoted as S and an intermediate representation as I. We shall denote any prefix by $\lhd$, any fragment by $\Box$, and any postfix by $\rhd$.

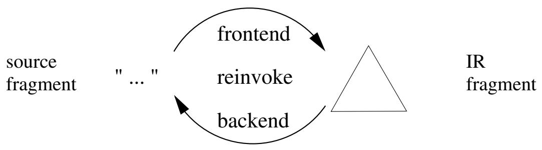


Fig. 2. A source-fragment can always be translated into an IR-fragment by invoking the frontend and an IR-fragment can always be translated into a source-fragment by invoking the backend

A given source-fragment, $S_{\square}$, is translated to an IR-fragment, $I_{\square}$, by invoking the frontend.

The fragment concatenator concatenates the source-prefix $S_{\triangleleft}$, the source-fragment $S_{\square}$, and the source-postfix $S_{\triangleright}$. Information necessary to extract the IR-fragment, $I_{\square}$, corresponding to the source-fragment, $S_{\square}$, from the IR of the completed program, shall be denoted L_{sep}. It represents separators that are inserted by the concatenator before invoking the frontend, and used by the extractor to separate the fragment from the prefix and postfix.

$$(S, L_{sep}) = \text{concatenator}(S_{\triangleleft}, S_{\square}, S_{\triangleright})$$

The completed program S can be parsed by the frontend

$$I = \text{frontend}(S)$$

to obtain the program in intermediate representation I. From this program I, the IR-fragment, $I_{\square}$, is extracted by the fragment extractor.

$$I_{\square} = \text{extractor}(I, L_{sep})$$

The fragment extractor strips off the IR-prefix, $I_{\triangleleft}$, corresponding to $S_{\triangleleft}$ and $I_{\triangleright}$ corresponding to $S_{\triangleright}$. Information on where these parts are separated, L_{sep}, which is returned by the fragment concatenator, is used to find start and end points of $I_{\triangleleft}$ and $I_{\triangleright}$.

We have shown how we can obtain the corresponding IR-fragment $I_{\square}$ for a given source-fragment $S_{\square}$ by invoking the frontend. The inverse operation, by invoking the backend, is

$$S_{\square} = \text{backend}(I_{\square}).$$

Since both representations, $I_{\square}$ and $S_{\square}$, can always be translated one to the other, both can be used interchangeably in the definition of a transformation.

In Fig. 2 this correspondence is shown as a diagram. The definition of a transformation is simplified because source-fragments, $S_{\square}$, can be used to define source code patterns as strings. On the other hand, source-fragments corresponding to subtrees of the IR can always be used as values in an attribute evaluation because we can always obtain the corresponding source-fragment for an IR-fragment.

This allows the definition of a transformation by combining sequential strings although the intermediate representation has a tree structure. All semantic information, such as type information for each expression, symbol tables, etc., is updated by the underlying system. Note that the order in which the IR is processed is (mostly) source sequence.

2.2 Fragment Substitution

An IR-fragment, $I_\square$, which is obtained from the fragment extractor, substitutes an IR-fragment in the IR as specified by L_{abs}.

$$I^{i+1} = \theta(\langle I^i_\square, I_\square, L_{abs}\rangle, I^i)$$

The substitution θ replaces the IR-fragment $I^i_\square$ by the new IR-fragment $I_\square$ (which corresponds to a $S_\square$) at the specified location L_{abs}. $I^i_\square$ is an IR-fragment in I^i. After a substitution has been applied the restructured IR, I^{i+1}, becomes accessible for the next transformation. This ensures that a substitution operates as a side-effect free function, with respect to the IR structure, in a transformation.

Once an attribute evaluation has been performed and a transformation is finished, I^{i+1} becomes accessible and I^i is no longer accessible. Note that fragments $I^i_\square$ and $I_\square$ can be empty, corresponding to empty strings ϵ (source-fragments), which allows to define insertions and deletions.

3 Program Transformations

Program transformations are specified as semantic actions of the abstract C++ grammar. The abstract grammar covers full C++. We use a successor of Coco/R [4], the C/C++ version ported by Frankie Arzu. Coco/R is a compiler generator that allows to specify a scanner and a parser in EBNF for context free languages. The grammar has to be LL(1). We use this tool to operate on the token stream of AST nodes. Therefore we do not use the scanner generator capabilities of Coco/R and implemented a scanner to operate on a token stream of AST nodes.

A terminal in our default abstract grammar always directly corresponds to AST nodes of one type. The name of this type is the name of the terminal in the grammar. The grammar can be modified but the user has to ensure that it still accepts all programs that are to be transformed. Our present version of the default abstract grammar for full C++ has 165 rules. Non-terminals either directly match names of base types in the AST's object-oriented class hierarchy, or the non-terminals were introduced (with the postfix NT in our default grammar) for better readability. The user can also access all annotated AST information gathered by the frontend at each AST node through a variable `astNode`. The variable always holds the pointer to the corresponding AST node of a parsed terminal.

In the example source in Fig. 3 we show an iteration on a user-defined container with an iterator. This pattern is frequently used in applications using

Before Transformation

```
for(ValContainer::iterator i=l.begin(); i!=l.end(); i++) {
   a.update(*i);
}
```

After transformation

```
#pragma omp parallel for
for(int i = 0; i < l.size(); i++) {
  a.update(l[i]);
}
```

Fig. 3. An iteration on a user-defined container `l` that provides an iterator interface. The object `a` is an instance of the user-defined class `Range`. Object `l` is of type `ValContainer`. In the optimization the iterator is replaced by code conforming to the required canonical form of an OpenMP parallel for. The user-defined method update is thread-safe. This semantic information is used in the transformation

C++98 standard container classes. The object `a` is an instance of the user-defined class `Range`. The transformation we present takes into account the semantics of the type `ValContainer` and the semantics of class `Range`. The transformation is therefore specific to these classes and its semantics.

For the type `ValContainer` we know that the type `iterator` defined in the class follows the iterator pattern as used in the C++98 standard library. For the type `Range` we know that the method `update` is thread safe. We show the core of a transformation to transform the code into the canonical form of a for-loop as required by the OpenMP standard. We also introduce the OpenMP pragma directive. Note that the variable `i` in the transformed code is implicitly private according to the OpenMP standard 2.0 . If the generated code is compiled with an OpenMP compiler, different threads are used for executing the body of the for-loop. The test, `isUserDefIteratorForStatement`, to determine whether the transformation can be applied, is conservative. It might not always allow to perform the optimization although it would be correct but it is never applied when we cannot ensure that the transformed code would be correct.

In the example in Fig. 4 the grammar rule of `SgScopeStatement` is shown. The terminal `SgForStatement` corresponds to an AST node of type `SgForStatement`. The variable `astNode` is a pointer to the respective AST node of the terminal and assigned by our supporting system when the scanner accesses the token stream. Note that every terminal in the grammar corresponds to a node in the AST, except the parentheses.

Methods of the object `subst` allow to insert new source code and delete subtrees in the AST. The substitution object `subst` buffers pairs of target location and string. The substitution is not performed before the semantic actions of all subtrees of the target location node have been performed. This mechanism allows to check whether substitutions would operate on overlapping subtrees of

```
SgScopeStatement<bool isOmpFor>
  = SgForStatement
      (.
        isOmpFor
        = ompTransUtil.isUserDefIteratorForStatement(astNode,isOmpFor);
      .)
    "(" SgForInitStatementNT<isOmpFor> SgExpressionRootNT
        SgExpressionRootNT SgBasicBlockNT<isOmpFor>
    ")"
      (.
        if(isOmpFor) {
          string ivarName = query.iteratorVariableName(astNode);
          string icontName = query.iteratorContainerName(astNode);
          string modifiedBodyString
                 = ompTransUtil.derefToIndexBody(ivarName,icontName);
          string beforeForStmt
                 = "#pragma omp parallel for\n";
          string newForStmt = "for( int "+ivarName+"=0;"
                            + ivarName+"<"+icontName+".size();"
                            + ivarName+"++ ) "+modifiedBodyString;
          subst.replace(astNode,beforeForStmt + newForStmt);
        }
      .)
  | ...
```

Fig. 4. A part of the grammar rule of SgScopeStatement of the abstract C++ grammar with the semantic action specifying the transformation of a SgForStatement

the AST (in the same attribute evaluation). In case of overlapping subtrees an error is reported.

The object `query` is of type `AstQuery` and provides frequently used methods for obtaining information stored in annotations of the AST. These methods are also implemented as attribute evaluations.

The inherited attribute `isOmpFor` is used to handle the nesting of for-loops. It depends on how an OpenMP compiler supports nested parallelism whether we want to parallelize inner for statements or only the outer for statement. In future this decision will be made more specific to OpenMP compilers on different platforms and the boolean attribute will be replaced by an object to provide more information about the context of OpenMP for-loops.

The object `query` of type `AstQuery` offers methods to provide information on subtrees that have been proven to be useful in different transformations. In the example we use it to obtain the name of the iterator variable, and to obtain the node of the declaration of the iterator variable. Note that these functions must return valid values because it has been tested that the for-loop qualifies for transformation before.

The example shows how we can decompose different aspects of a transformation into separate attribute evaluations. The methods of the query object are implemented by using the attribute evaluation. For this reason we allow to call any method of the recursive descent parser generated by COCO to parse a sublanguage, and start an evaluation at a certain node in the AST. Multiple grammar files can also be used for such cases and each file contains a version of the abstract C++ grammar. In the example, `isUserDefIteratorForStatement` is a wrapper function of another attribute evaluation generated by COCO that starts at a SgForStatement node.

In Fig. 3 the generated code is shown. The access uses the notation for random access iterators. Even if the access is not of complexity O(1) the parallelization can still provide speedup. The user who implements the transformation has to take such tradeoffs into account in a test function to decide whether a transformation should be applied or not. Note that the generated source code can have a slightly different formatting because the format of the source code is a beautified version of the source corresponding to the transformed AST.

4 Related Work

We use Sage III as intermediate representation, which we have developed as a revision of the Sage II [5] AST restructuring tool. Its predecessor, Sage++, included a Fortran frontend, while Sage II included the EDG C++ frontend [3] and represented a more robust handling of C++ as a direct result. Our work has substantially modified Sage II (e.g., adding template support and changes of the structure and interfaces of Sage II of about 25% of the node classes). Sage II required modifying the AST by explicitly rearranging pointers between AST nodes and creating new node objects if new code needed to be added. In our framework this can be done by using source strings and an abstract grammar.

Related work on the optimization of libraries on telescoping languages [6] shares many of the same goals as our research work and we expect to work more closely with these researchers in the near future. Our approach so far is less ambitious than the telescoping languages research, but is in some aspects further along, though currently specific to abstractions represented in C++.

Further approaches are based on the definition of library-specific annotation languages to guide optimizing source code transformations [7] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations, see [8] for example. We address the need of annotations for guiding optimizations either by pragmas, comments, or make optimizations specific to user-defined types as discussed in the example transformation.

Kimwitu [9] allows to associate semantic actions with rules of a tree grammar. Conceptually Kimwitu could be used instead of COCO as well. But the substitution mechanism is more difficult to integrate into our system when using the C++ version of Kimwitu from our experience because it uses its own memory handling and puts restrictions on some code fragments used in semantic actions. COCO was easier to integrate in our system because it only copes with issues of parsing and not transformation, and does not put any restrictions on the code used in semantic actions. However, our grammar conforms to the essential properties of a tree grammar as required by Kimwitu. The other mode of Kimwitu, to express term rewriting explicitly by using subterms describing subtrees on both sides of a rule, is an advantage of Kimwitu in particular for the compact specification of algebraic optimizations.

The Microsoft .NET CodeDom Compiler Framework is used by various tools, including ASP.NET and Visual Studio.NET. It offers an interface for restructuring source-code, designed to handle different languages. Nigel Perry has defined an abstract grammar for the CodeDom Language [10]. A program in the language is represented by a tree of CodeDom objects, which corresponds to a parse tree in a compiler for a conventional language. In our framework the abstract grammar can actually be used to specify transformations. In the abstract CodeDom EBNF grammar, type information is made explicit as extension in the grammar. In our grammar type information is available as accessible annotation of the AST nodes. Also we do not use tree extensions to identify grammar symbols that correspond to AST nodes. A terminal always directly corresponds to an AST node. We only added parentheses to the token stream. However, most of all information required for our approach is available for the CodeDom framework, which makes it an interesting target in our future work.

5 Conclusions and Future Work

The use of an abstract grammar greatly simplifies the specification of a source-to-source transformation. Many aspects of parsing source code and type evaluation are not helpful for expressing code transformations. The specification of a source-to-source transformation should not interfere with specific parsing issues of the concrete syntax of the language. On the other hand, the concrete syntax is what developers, who want to optimize their application codes, are most familiar with. From this we conclude that offering the use of source-strings for specifying new code and using an abstract grammar to allow to specify transformations is a practical solution to this problem. The availability of full type information is necessary for the optimization of user-defined abstractions.

Instead of requiring the user to learn a new language to express transformations, all transformations are themselves defined in C++, the same language in which the application code is written and which the user seeks to optimize. The grammar forces the user to structure the transformation according to the structure of the language, the decomposition in different transformation objects, as shown in the example, gives the necessary freedom in designing complex transformations.

Future work is targeted at demonstrating the development of a wide range of optimizing source-to-source translators for specific scientific libraries and applications. Additional work is the analysis of complex data structures to automate the generation of application specific tools (connection to visualization libraries, dump/restart functions, etc.).

By permitting developers to add highly tailored companion optimizations to their user-defined types and applications, we define a hierarchical (telescoping) approach to language design which builds incrementally upon existing general purpose languages. We hope that a similar approach could in the future form a significant mechanism within a general purpose language compiler to allow users to extend the range of optimizations.

References

1. Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *16th International Parallel and Distributed Processing Symposium (IPDPS, IPPS, SPDP)*, pages 105–114. IEEE, April 2002.
2. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency: Practice and Experience*, to appear.
3. Edison Design Group. http://www.edg.com.
4. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
5. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.
6. B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, K. Kennedy, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. In *Journal of Parallel and Distributed Computing*, 2000.
7. Samuel Z. Guyer and Calvin Liri. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, Berkeley, CA, October 3–5 1999. USENIX Association.
8. Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Conference Proceedings of the 1999 International Conference on Supercomputing*, pages 434–443, Rhodes, Greece, June 20–25, 1999. ACM SIGARCH.
9. P. van Eijk, A. Belinfante, H. Eertink, and H. Albas. The term processor Kimwitu. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 96–111, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.
10. Nigel Perry. A definition of the codedom abstract language, http://www.mondrian-script.org/codedom/codedom grammar.html, 2002.

An Oberon Linker for an Imperfect World – More Notes on Building Your Own Tools

Paul Reed

Padded Cell Software Ltd, PO Box 1880, London NW6 1BQ, United Kingdom
paulreed@paddedcell.com
http://www.paddedcell.com

Abstract. Experience creating custom application software has taught us that total control over our development tools is a necessity. Project Oberon provided an excellent starting point for us to build our own cross-platform application programming environment. In addition to adapting Wirth's Oberon compiler, we have developed an accompanying linker which creates native programs for popular operating systems such as Windows, MS-DOS, Unix/Linux, Macintosh, and Palm OS. In this case study, we summarise the structure of the linker and some of the executable formats it can generate, and we describe one of the large commercial projects in which it is used. Note that such a linker is not necessary in the Oberon operating system, since the system loads compiled modules directly into memory as needed; sadly many industry-standard operating systems are not as efficient.

1 Introduction and Motivation

Established in 1988, Padded Cell Software Ltd specialises in building custom software systems for small to medium-sized businesses. Since 1997, we have been writing commercial-grade application software in the Oberon programming language, using our own compiler [1] adapted directly from Project Oberon [2].

Before we discovered Oberon, we had become very dissatisfied with development tools available commercially, which were complex and bug-ridden. The programs we created using them contained large, intricate libraries which often exhibited unexpected behaviour at critical points in a project, and the programs also suffered from delicate dependencies on other software installed on an end-user's machine. Under the tight constraints of time and budget that apply in the real world, such problems can threaten the viability of an entire project.

The trend toward greater complexity (and therefore bug-count) seems to be continuing; Sun's Java 2 (version 1.4.1) needs 120MB of disk space on a development machine, and an 8MB run-time environment on each end-user machine; Microsoft's .NET (version 1.1) requires 850MB for development, and a 23MB run-time. We remain unconvinced that this bulk, which also characterises most end-user programs [3] as well as development tools, is inevitable. Our compiler and linker together comprise only around 250KB. Many of the end-user programs we produce are less than 100KB (complete, no run-time required).

This case study describes our Oberon linker, which creates native programs for the operating systems in use by our customers, the majority being Microsoft Windows, Apple's MacOS for the Macintosh, and Unix/Linux-type systems.

The linker takes object files (produced by the compiler from a module's source code) and generates a single executable file in the format required by the target operating system. Note that by contrast, in the carefully-engineered operating system described in Project Oberon the additional step of linking is unnecessary. Object files are directly loaded (once) into memory when required, leading to extremely compact use of resources.

To complete the case study, the varied executable formats which the linker produces are summarised, and a large commercial project is discussed during which the linker was used and enhanced.

2 Linker Structure

An executable file is created by the linker in two phases. First it is necessary to decide, independently of the particular target format desired, which modules will be required to provide object files for the final executable file. The Oberon language construct which provides this information is the `IMPORT` statement at the beginning of each module. Typically, a module will use (i.e., import) functionality defined and implemented in several other modules, which in turn import further modules. The user gives the name of one base module (usually a main program or event loop) to the linker via its command-line, and all other modules required are then discovered recursively by following the import relationships. As each module's object file is scanned by the linker, a list is formed containing information about each module, such as code size and the required size of module variables.

In the original Oberon system, the lowest-level modules (those which import no others) are device drivers and the kernel of the operating system. In our case, library modules import functionality from the target operating system using the Oberon `PROCEDURE-` construct. Here is an example from a Windows module:

```
PROCEDURE- ShowWindow(hWnd: HANDLE; nCmdShow: LONGINT); "USER32";
```

Once this has been defined (with no procedure body), it can be called like any other Oberon procedure. The external procedure name is stored by the compiler in the object file, ready to be placed by the linker in a special section of the executable file (see below).

During the second phase, the list of module information is scanned with a particular target in mind, creating a complete picture of the final locations of the code and the pre-initialised data (constants) from each of the modules in the output file. Additional data structures in the output file required by the target operating system are taken into account. The entire executable file can now be written efficiently in one pass, transferring the code and constants from each object file to their correct place in the output file.

The simplicity of this linker is due in part to the way in which the Oberon language and the compiler handle cross-module references, using what is called *separate compilation.* Symbols exported by a module are kept in a separate symbol file, not in the object file. This helps the compiler support strong-typing across module boundaries (enforcing language rules), but it has a useful side-effect for our purposes: exported variables are referred to by (relative) address, and exported procedures by procedure number. This is perfectly adequate information for the linker, so that unlike linkers for other languages such as C, it is not necessary to resolve textual symbols. Symbol files are not needed at all during the linking process.

3 Executable Formats

The MS-DOS EXE file [4] is probably the most studied executable format, due to MS-DOS's popularity amongst users and programmers since its debut in 1981. The EXE file consists of a small header containing file statistics (sizes, entry point, etc.), an optional fixup list, and then the code and pre-initialised data of the program. *Fixups* highlight machine instructions within the program which need to be altered to take account of the actual address at which it is loaded; this is not normally known at compile- or link-time.

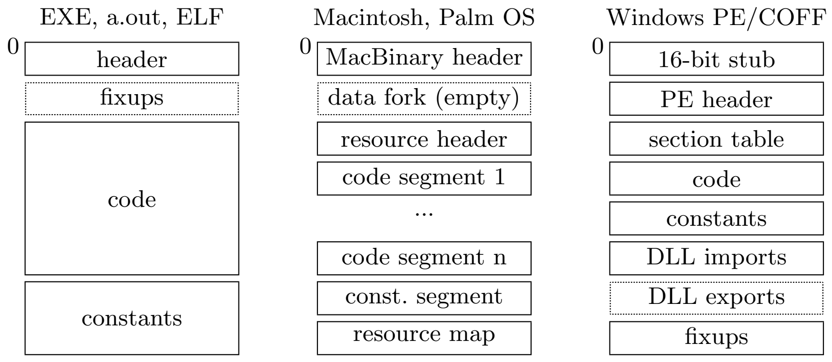


Fig. 1. Styles of executable file format (start of files at the top)

Similarly, a Unix/Linux a.out format file [5] has a small header, followed by a block of code and also constants which are loaded together into memory. The Unix/Linux ELF format [6] is slightly more complicated, since it contains extra headers and tables to support its use as an intermediate object file format as well as an executable format. Both formats contain padding bytes, used to align sections on page boundaries, where a page is the smallest unit of memory (often 4KB) managed by the Unix/Linux virtual memory system.

A Macintosh program is composed of several *segments* [7], each of which is loaded on demand (and purged from memory when no longer needed) resulting in better memory use on small machines. Every Oberon module is mapped to a separate segment for simplicity, with a final segment containing the constants from all of the modules. Each segment is held in the file as a Macintosh resource, the locations of the resources being stored in a *resource map* at the end of the file. Uniquely, Macintosh files have two separate *forks*, one for data (often empty in a program file) and one for resources. To handle this the MacBinary [8] interchange format is used when cross-linking from other systems. The ability of the linker to pre-calculate the sizes of all the objects in the output file (by traversing the list of modules in memory) allows an entire MacBinary file to be written in one pass, starting with the MacBinary header, continuing with the actual code resources, and finishing with a resource map.

The Palm OS executable format [9] uses the same concept of storing code in resources, but with a simpler resource map (at the beginning rather than at the end of the file) and without the complication of a separate data and resource fork.

The Microsoft Windows Portable Executable and Common Object File Format (PE/COFF) [10] is the most complicated of the executable formats which our linker has yet supported. The file is split into many sections, each of which has a particular function, and which may or may not be mapped into memory in pages as in Unix/Linux. It is an EXE-file stub which begins the file, executed when the PE/COFF file is run in a 16-bit environment such as MS-DOS. At the start of the PE/COFF file proper is a general header describing the file, followed by a table describing the sections. One section contains all of the code, and another contains the constants. A Dynamic-Link Library (DLL) imports section describes all the routines required by the program from external DLLs, including system DLLs such as KERNEL32.DLL. A similar optional section includes functions exported by the program when creating a DLL.

The final PE/COFF section written by the linker is entirely devoted to fixups, which are grouped by the pages that they refer to. This is an artefact of the way that pages from the executable file are mapped into memory on demand. However, the linker is not in a position to write the file until it has discovered how large the fixup section needs to be, because the size will be placed in the fixup entry in the section table. A great deal of information has to be gathered, by traversing the linked list of modules several times, before writing anything. This part of the linker was by far the hardest to get right.

4 Building www.rexfeatures.com

Rex Features Ltd., a picture library and press agency, is one of the UK's largest with offices in London and New York, and agents and photographers world-wide. Late in 2000 they commissioned us to build a website which would allow their archive of more than a third of a million digital images to be searched on-line for selection and download. A large part of Rex's business is news-releated, so speedy delivery of images (under password control) via the web would increase their competitiveness, as well as being more cost-effective than the existing methods of ISDN, email, and motorbike courier.

Using a team of two programmers (the author and a Rex staff member) the main search functionality was built using Common Gateway Interface (CGI) programs written in Oberon. A prototype was developed using Windows NT as the server platform. However, security and reliability concerns led to a decision to run the final website on a Unix/Linux-type operating system — ultimately FreeBSD was chosen. This required us to enhance our linker mid-project to produce the ELF executables described above, and new library routines were needed to target the new operating system.

The final website was begun in earnest in the last quarter of 2001. Development proceeded smoothly, and the website went live as scheduled in April 2002. In the six months to October 2002, the webserver received a total of 257 million hits, sometimes exceeding 300,000 CGI program executions in a single day.

5 Conclusion

Building our own lean development tools based on the Oberon project continues to reap dividends. We create more efficient, compact and robust commercial custom software systems, and in more predictable timescales, than with the unwieldy commercially-available development environments we used to use.

Our Oberon linker produces native executable binaries for a wide variety of operating systems other than the Oberon system. A target-independent first phase recursively builds a list describing the modules to be linked, while a target-dependent second phase traverses the list producing the output file in one pass. Oberon's concept of separate compilation, which stores symbol information and object code separately during a compile, reduces considerably the work to be carried out by the linker.

References

1. Reed, P. (2000). Building Your Own Tools - An Oberon Industrial Case Study, in Modular Programming Languages - Lecture Notes in Comp. Science No. 1897. Springer.
2. Wirth, N. and Gutknecht, J. (1992). Project Oberon: The Design of an Operating System and Compiler. Addison Wesley.
3. Wirth, N. (1995). A Plea for Lean Software. IEEE Computer, vol. 28, no. 2.
4. Burgoyne, K. Article 4 – Structure of an Application Program, in Duncan, R. (Ed.) (1988). MS-DOS Encyclopedia. Microsoft Press
5. OpenBSD Man pages, `http://www.openbsd.org`.
6. Executable and Linkable Format - Tools Interface Standards, Portable Formats Specification, Version 1.1. UNIX Systems Laboratories
7. Apple Computer (1985). Inside Macintosh. Addison Wesley
8. Brothers, D.F. (1985). Macintosh Binary Transfer Format "MacBinary" Standard Proposal. Micro-networked Apple User's Group
9. Introduction to File Formats (2002). PalmSource Inc. `http://www.palmos.com`.
10. Microsoft Portable Executable and Common Object File Format Specification. Microsoft Corporation. `http://www.microsoft.com`.

Language Definition in the Schütz Semantic Editor

Rodney M. Bates

Department of Computer Science, Wichita State University
1845 Fairmount, Wichita, Kansas, 67260-0083, USA
rodney.bates@wichita.edu

Abstract. The Schütz semantic editor intends to perform semantic analysis, incrementally during editing, on possibly incomplete and incorrect programs. Currently, the project is addressing a prerequisite problem of maintaining an internal tree representation, necessary for semantic analysis, while presenting the user with a text-editor style interface, essential for programmer acceptance. Prior papers have addressed the language-independent aspects of our data structure and algorithms. Here, we discuss our specifications of the language to be edited. These are initially written in a purpose-designed Language Definition Language (LDL), then transformed into internal form needed by the editor. Our approach is to first build a syntactic editor for its own LDL, then use this to define an editor for its own implementation language, i.e., Modula-3, add incremental semantics, and finally to define editors for other programming languages. We summarize our language-independent data structure and algorithms and how they are specialized for a specific language. We describe our LDL in detail and how it has evolved to support Modula-3. The current implementation supports editing of LDL descriptions. The principal contribution of this paper is that LDL is an essential part of our method of providing an acceptable user-interface, without which semantic editing would be irrelevant.

1 Introduction

Over a decade ago, there was widespread research activity in syntax-directed editors for programming languages. It is an understatement to say that users were not pleased with these, primarily because they present a user interface that is very hard to use. Conventional text editors, e.g. [9] allow users to accomplish their editing goals far more quickly.

Although syntax-directed editing was rejected by users, it has significant unrealized potential in other ways. Since it maintains a form of syntactically analyzed code, it can be the basis for many language-aware browsing and editing functions, especially static semantics. Several programmers' tools have been developed which do at least simple forms of semantic browsing. These also have not realized the potential of semantic editing, because they only work on code that is complete and statically correct, or nearly so. This is usually unrealistic.

Several semantic editors were developed at about the same time [1][14][11]. These provide incremental static semantic analysis and work on incomplete and incorrect programs. The PSG system [1] can, for example, infer or partially infer declarations, types, etc., from uses of identifiers. These generally have at least partially syntax-directed front ends also.

Internal representations in parsed form are chronically large. They commonly run an order of magnitude larger than textual files. With increasing interest in whole-program analysis and optimization, working sets for these representations can easily exceed even the large, cheap memories found on today's computers.

The motive for this work is the development of a semantic editor, which we call Schütz, that can offer the kinds of semantic functions we have described, work on incomplete and incorrect programs, and simultaneously present a true text-editor style user interface. Such a tool needs an internal syntactic representation of the edited programs, which we also wish to reduce in size.

Although semantic editing is the ultimate goal of the project, much of our work to date has addressed the prerequisite goal of presenting an acceptable user interface, while maintaining a syntactically analyzed internal representation on which semantic analysis can be built. Without this, a semantic editor would be rejected, as were syntax-directed editors. The primary contribution of this paper is our method of defining the syntax of language to be edited, in such a way that a text-editor style user interface can be provided.

The central representation and algorithms are language-independent. A language definition is expressed in a declarative notation, creatively named *Language Definition Language* (abbreviated *LDL*) and converted mechanically to the tables and data structures that specialize Schütz for a specific language. Here, we summarize our language-independent data structure and algorithms, as reported more fully in two previous papers, ([3] and [5]) then describe our LDL in some detail and how it has evolved to support Modula-3.

The remainder of this paper is organized as follows. Section 2 describes the relation of this work to other language-aware editors. Section 3 briefly summarizes our internal data structure. Section 4 describes LDL, with subsections on its main parts. Section 5 describes some modifications to LDL that were motived by its use for Modula-3. Finally, in Sect. 6, we mention additional work on LDL that the Schütz project requires.

Technical reports describing the LDL in more detail and the process of bootstrapping Schütz, as well as LDL definitions of LDL and Modula-3, are available at the author's web pages, at `http://www.cs.wichita.edu/~bates/scheutz`.

2 Relation to Other Work

There are many editors with degrees of internal knowledge of a programming language. For a representative sampling, see [7][8][10][12][13][14]. The *template editors* generally maintain a textual representation, but can insert predefined blocks of text called *templates*. The *syntax directed editors* maintain a parsed representation. They allow code modifications using knowledge of the syntactic structure.

The PSG system [1] maintains a tree representation, but allows that any subtree, at user request, may be unparsed, i.e. converted to textual form, after which text-editing may be done within. Reparsing of this region is done later, also at user request. The syntactic consequences of text editing cannot propagate out of the region. Our scheme does not have this restriction, nor does it require the user to explicitly specify a region to be unparsed. Our design of LDL and our intended semantic analysis owe much to PSG. In [13], Reiss describes the PECAN system as allowing both template and text-oriented editing, but does not describe the way these are integrated and does not describe a representation or algorithms for doing so.

The recent and extensive Harmonia project [7] is the closest to this work, sharing many of our goals. Our representation is different and reflects our concern about size. Although too little information about either system is available, we estimate our representation may be about half as large as Harmonia's.

3 Summary of Text Syntax Trees

In this section, we summarize the internal representation used in Schütz, which we call a *Text Syntax Tree (TST).* A more complete descriptive explanation can be found in [3] and a formal description in [5].

A single TST represents a program or fragment being edited, including non-syntactic content. A leaf node is a terminal with multiple spellings or *lexemes.* In a typical language, this would include the identifiers and the literals. An interior node contains an abstract token and a sequence of zero or more *children.* Each child is a TST subtree, along with additional information used in navigation and layout.

In the usual abstract syntax tree (*AST*) representation (see, e.g. [2, pp286-292]) each tree node has a distinct *abstract symbol*, which denotes a production rather than a nonterminal and calls for a distinct, fixed list of children. The children are usually represented as fields that point to the objects representing the children. The list of children is different for each symbol. For example, the AST node for a variable declaration might have three children: the variable name, its type, and its initial value. In a TST, each node has a symbol, as above, but the list of children is variable in length. Some of these, called *AST children*, are the same children the node would have in a conventional AST, and these are always present. As with a conventional AST, most of the tokens of the concrete syntax are absent.

Each abstract symbol has a corresponding *format syntax tree*, abbreviated FST, whose root also contains this symbol. Usually, an FST will have only a root and a few leaves as its immediate children. The leaves denote, left-to-right, the lexical tokens needed to reconstruct the textual form of the AST node. Some FST children, called *insertion tokens*, denote the omitted delimiters. Others, called *AST children*, denote where the textual expansion of an AST child should be inserted.

A typical language has several kinds of lists, of variable length, of items all derived from the same nonterminal. Lists of declarations, formal parameters, etc., are common examples. Traditional AST representations often denote these lists using a node child that amounts to a sibling link pointer. This idiom does not require any special treatment in the form of the tree grammar. In Schütz, we treat lists specially, to support our method of incremental reparsing. Some interior TST nodes, distinguished by their abstract symbol and called *list nodes*, have a variable number of AST children, all in the same class. The FST corresponding to a list TST node always has exactly one AST child node. Insertion tokens following it denote the separators between list elements.

FSTs also define layout. Other leaf nodes give the location of line breaks and indentations. These are optionally applied, depending on static layout properties of the FST and dynamically computed information in the TST. There are FST subtrees that conditionally add insertion tokens, for example, extra parentheses around a subtree that is a list with plural elements.

Interspersed with the AST children of a TST node are *modifiers* of several kinds. These represent nonsyntactic information such as unparsed text, error messages, comments, syntax error repairs proposed by Schütz but not yet accepted by the programmer, etc. They are an important part of the design, but are language-independent, thus not relevant to LDL.

In most cases, the list of children of a TST node will be small. However, it could grow quite long, especially in the case of a list node, as the source module being edited grows. Since we wish to use a TST in an editor, with a human interacting, we wish to minimize the variation in computation time from one interaction to another, although this need not be extremely short. To this end, we use a representation of the child list of a TST node called a *sequence tree*. The sequence tree is described elsewhere in detail[4][6]. It allows slicing and concatenation of arbitrary subsequences of a list of children to be done in time logarithmic in the length of the child list. It also supports logarithmic time access to individual children.

Figure 1 shows an example of TST representation and its expansion. At the top is the represented text, in Modula-3. Note that the reserved work `THEN` is absent. It has been deleted by the user's text editing actions. The top tree is the corresponding TST. The two middle children of its root are modifiers, one to explicitly delete `THEN`, which is normally implicitly present, and one to insert a comment.

The two trees shown side-by-side are the relevant FSTs for the AS nodes in the example. The numbers beside the arcs are used to associate FS children with TST children. Asterisks denote AST children. At the bottom is a conceptual, composite tree, constructed by inserting copies of FSTs in place of abstract nodes in the TST. Its leaves denote the represented text. This tree is never constructed explicitly in Schütz, but is conceptually traversed by traversing the TST in concert with the FSTs of the language.

4 The Language Definition Language

In this section, we describe Schütz' *Language Definition Language (LDL).* LDL is a textual notation that gives properties of a specific language. It comprises

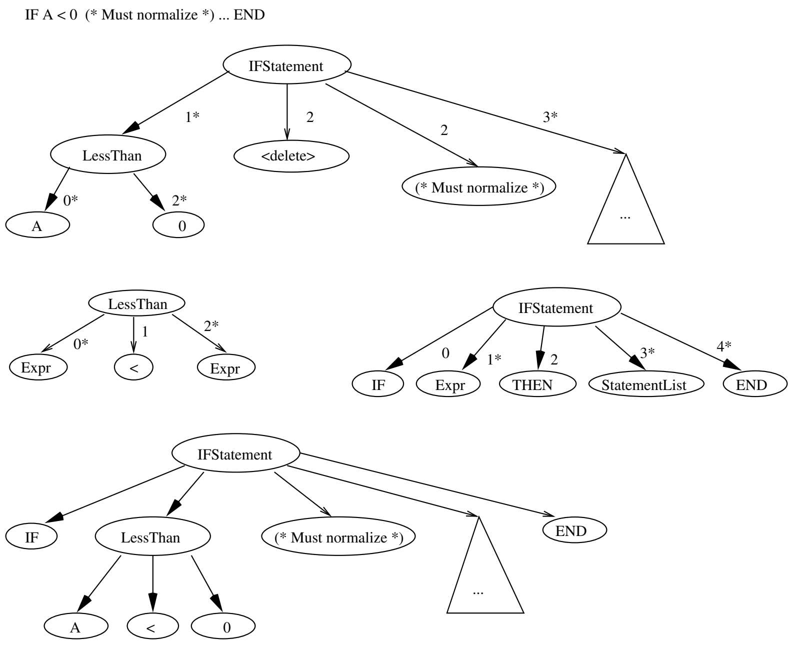


Fig. 1. TST example

primarily an Abstract Syntax (AS), a Concrete Syntax (CS) and a Format Syntax (FS). It additionally contains some other definitions, e.g. of the grammar start symbol and terminals with variable spelling, such as identifiers.

Terminals with fixed spelling do not appear in ASTs, but are needed in both the CS and the FS. Leaves of an AST are *abstract terminals* and have variable spelling, e.g. identifiers. Some nonterminals are used only temporarily in parsing. Others serve both in parsing and as tokens in interior AST nodes. These are called *abstract nonterminals.* Abstract nonterminals are further subdivided into *fixed* and *list* tokens, which appear in fixed and list nodes, respectively. LDL also uses *class* tokens, which do not appear in trees. They denote subsets of the tokens in the language. They may serve as non-abstract nonterminals during parsing.

The AS is a tree grammar for the AST core of TSTs. A *class rule* defines a class name by enumerating tokens that are its members. A class can be a member of another class, in which case, the token set of the inner class is flattened into the outer class. Class members can be any kind of token, but each member must be meaningful in each context where the class name is used.

A *fixed rule* defines a fixed, abstract nonterminal. AST nodes with this token have a fixed list of children, each possibly of a different class. A *list rule* defines a list, abstract nonterminal. AST nodes containing this token have a variable number of children, each in the same class.

The CS is a more-or-less conventional context free grammar, used for parsing. If the LHS nonterminal of a CS rule is elsewhere declared as an abstract nonterminal, then an AST node containing that nonterminal is built whenever the rule is reduced. A CS rule can also separately specify an abstract nonterminal, different from the rule's LHS, to be built when the rule is reduced. Otherwise, a CS rule does not build an AST node, but depends on reductions higher in the derivation tree to do the building.

The FS gives rules for rebuilding a textual notation from an AST. Each abstract nonterminal has an FS rule whose RHS repeats the children of the AS node, but intersperses with them, insertion tokens and formatting directives to control layout. FS rules express the same kind of layout possibilities as FSTs. They can have multi-level conditional layout specifications and can also specify conditional insertion of fixed terminals that depends on properties of AST subtrees. These are used, for example, to reinsert necessary parentheses into expressions, required by the precedence rules of the CS, but absent in the AS.

4.1 The Concrete Syntax

The metalanguage we use to describe the syntax of LDL is identical to LDL's own CS notation. We have written a complete description of LDL in LDL, but it is confusing, both to write and to read. It is not a very practical way to initially define LDL, because it amounts to a large system of equations in some unidentified algebra, whose solution is the language we want to define. Instead, we give the customary informal description.

Strings enclosed in double quotes are literal terminals. "`Ident`", "`String`", and "`Integer`" are terminals of variable spelling, with the usual meanings. Other strings of letters are nonterminals, except for certain LDL *reserved words*, which are spelled in all capitals. "`::=`" means produces. Each production ends with a period. Alternation is denoted by "`|`". Braces "`{`" and "`}`" mean closure, i.e. zero or more occurrences of the material inside. If the matching close brace is "`}+`", it means one or more occurrences. When the symbol "`||`" occurs inside braces, e.g. "`{ A || B }`", it means `B` is a sequence of separator tokens between `A`'s, where `A` is any subproduction and `B` is one or more literal terminals. Material in brackets "`[`" and "`]`" is optional. We use the adjective "described" to denote things in the language being described, not in LDL itself.

4.2 The Abstract Syntax

An AS specification consists of *variable terminal rules*, *class rules*, *fixed rules* and *list rules*. A variable terminal rule just declares a described identifier as denoting a terminal with variable spelling. A class rule declares that its LHS identifier is the name of a class that contains an enumerated list of node kinds.

```
AsClassRule ::= Ident "=" Alternation "." .
Alternation ::= { CsAtom || "|" }+ .
CsAtom ::= Ident | String .
```

The example below declares that an `Expr` is either a `Plus`, `Times`, or a `Literal`. Each of the alternatives must be declared as some kind of AST node, or as another class. In the latter case, the subclass named in the RHS is flattened into the declared class, so that it directly includes the members of the subclass. Cyclic class declarations are illegal.

```
Expr = Plus | Times | Literal .
```

A fixed rule declares an identifier to be the name of a kind of AST *fixed node* and lists the classes and optionally the names of each of its children.

```
AsFixedRule ::= Ident ":=" AsChildList "." .
AsChildList ::= { AsChild || ";" } .
AsChild ::= AsReqdChild | AsOptChild .
AsReqdChild ::= [ Ident ":" ] Ident .
AsOptChild ::= [ Ident ":" ] "[" Ident "]" .
```

The two examples below declare two AST nodes, `Plus` and `Times`. Each has the same two children, each child being a member of class `Expr`. The names `Left` and `Right` identify the children. The name and following colon are optional. They have only minor importance in syntactic specifications but are needed for the semantic view of a TST as AST.

```
Plus := Left : Expr ; Right : Expr .
Times := Left : Expr ; Right : Expr .
```

A child of a fixed node can be optional, denoted by enclosing its class name in braces:

```
WhileStmt := Cond : Expr ; Body : Stmts .
IfStmt := Cond : Expr ; ThenClause : Stmts
         ; ElseClause : [ Stmts ] .
```

A list rule declares an AST *list node* with variable number of children, each a (possibly different) member of a specified class.

```
AsListRule ::= AsStarRule | AsPlusRule .
AsStarRule ::= Ident ":=" [ Ident ":" ] "{" Ident "}" "." .
AsPlusRule ::= Ident ":=" [ Ident ":" ] "{" Ident "}+" "." .
```

If the closing delimiter is "`}+`", then there will always be at least one child. Otherwise, there may be zero. The following example declares `StmtList` to be a list node, each of whose children is a `Stmt`.

```
Stmts := { Stmt } .
```

4.3 Format Syntax Rules

The FS specifies the mapping from ASTs back to textual notations. It reinserts the delimiter tokens that are omitted in the AST and gives layout rules for indentation, etc.

```
FsRule ::= FsFixedRule | FsListRule .
FsFixedRule ::= Ident "->" FmtKind FsFixedChildList "." .
FmtKind ::= [ "HORIZ" | "VERT" | "FILL'' ] .
```

The identifier in the LHS of a fixed FS rule must be declared elsewhere as an AST fixed node. FS rules specify that the layout should be horizontal, vertical, or filled. This property is called a *format kind.* `VERT` means that all line breaks are unconditionally taken, thus the construct always occupies multiple lines. `HORIZ`, the default, means that the line is formatted horizontally, i.e., none of the line breaks are to be taken, if the entire construct will fit on the line. Otherwise, it is formatted vertically. `FILL` specifies that the line breaks are treated independently of each other. Each will be taken only if the text up to the following line break or end of the construct fits on the current line. Vertical and horizontal formatting are possible special cases.

```
FsFixedChildList ::= { FsFixedChild } .
FsFixedChild ::= FsChild | LineBreak | String .
FsChild ::= FsChildPlain | FsSubtree | FsChildCondFmt .
FsChildPlain
  ::= [ Ident ":" ] [ "@" Integer ] IdentOrDontCare .
IdentOrDontCare ::= Ident | "_" .
```

The `FixedChildList` gives formatting information for the children of the AST node. A `String` is an insertion token. An `FsChildPlain` matches a child of the corresponding AST node kind. An AST node has only direct children, while an `FsRule` can have indirect descendents. The list of AST children is matched left-to-right to those leaves of the FS that are derived from `FsChildPlain`. It can name the child or class of the child or can be "`_`". The optional `Integer` is an additional indentation amount to be added to that inherited from the containing construct and applied to this entire construct.

```
LineBreak ::= "!" [ "@" Integer ] .
```

A `LineBreak` specifies that a new line is conditionally inserted here. If inserted, the new line begins indented by the dynamic indentation of the construct plus the value of the `Integer`, which defaults to zero.

```
WhileStmt -> HORIZ "WHILE" Expr "DO" ! @ 2 Stmts ! "END" .
```

The example rule above shows how the reserved words are reinserted surrounding the two AST children of a `WhileStmt` node. An example of text reconstructed by this rule, with vertical layout might be:

```
WHILE VariableWithLongName > LongNamedConstant DO
  ProcedureWhoseNameIsLong ( VariableWithLongName )
END
```

FS subtrees allow a subsequence of the children to have a different format kind than the parent. This works the same way as for a whole fixed FS rule, applied only to the subconstruct the `FsSubtree` matches.

```
FsSubtree ::= FmtKind "(" FsFixedChildList ")" .
FsChildCondFmt
  ::= FsCondPresent | FsCondNonempty | FsCondPlural
      | FsCondMember .
FsCondPresent ::= "PRESENT" "(" FsFixedChildList ")" .
FsCondNonempty ::= "NONEMPTY" "(" FsFixedChildList ")" .
FsCondPlural ::= "PLURAL" "(" FsFixedChildList ")" .
FsCondMember ::= "MEMBER" Ident "(" FsFixedChildList ")" .
```

The conditional formatting subtrees specify a list of inserted tokens, with exactly one AST child among them, that are inserted conditionally. If the condition is not satisfied, the construct is formatted as if the entire `FsFixedChildList` were replaced by only this AST child and flattened into the containing FS rule or subtree. If the condition is satisfied, the entire `FsFixedChildList` is thus-flattened.

The condition for `PRESENT` is that this child is present in the AST. The condition for `NONEMPTY` is that the child list is nonempty. The condition for `PLURAL` is that the child list contains at least two nodes. The condition for `MEMBER` is that the child in the AST corresponding to the `FsChild` is in the class `Ident`.

This example conditionally inserts an `ELSE` clause when it is present:

```
IfStmt
  -> HORIZ "IF" Expr "THEN" ! @ 2 ThenClause : Stmts
     PRESENT ( ! "ELSE" ! @ 2 ElseClause : Stmts )
     ! "END" .
```

The `MEMBER` condition is used to insert parentheses, as required to override normal precedence. For example, using the abstract rules above, the expression `A * ( B + C )` would typically be represented as an AST with a `Times` node on top and a `Plus` node as its right child. The parentheses are not present in the AST. They have to be inserted in the text, otherwise it would be rendered as `A * B + C`, which, by usual precedence rules, is a different expression. Here is an example FS rule for this:

```
Times -> MEMBER Plus ( "(" Expr ")" )
         "*" MEMBER Plus ( "(" Expr ")" ) .
```

FS list rules correspond to AS list nodes. The single AST child corresponds to the `FsChild`. As many occurrences of this child as the AST node has will be formatted, as for a child of a fixed node. If there are $N > 1$ list children, they will be separated by $N - 1$ copies of the formatting that the `FormatterList` specifies, as for a fixed node.

```
FsListRule
  ::= Ident "->" FmtKind
      "{" FsChild [ "||" FormatterList ] "}" "." .
FormatterList ::= { Formatter } .
Formatter ::= Ident | LineBreak | String .
```

For example,

```
Stmts -> { Stmt || ";" } .
```

might, for a list with three statements, give the textual result

```
I := 0 ; FuncCall ( Variable ) ; RETURN
```

4.4 Other Constructs

LDL contains other constructs that we mention only briefly. Like most LALR parser generators, it allows for precedence and associativity to be separately specified. Besides allowing for a simpler (but ambiguous) concrete grammar, this also allows the CS rules to more closely match the abstract and FS rules. The start symbol can be explicitly specified. The entire language definition is named and syntactically bracketed. Abstract terminals are explicitly declared.

5 Modifications for Modula-3

Our initial use of LDL has been to define an editor for itself, as part of a bootstrapping process. Once we began defining Modula-3 in LDL, we discovered some additional needs. While our initial LDL was indeed adequate, there were places where the FS did not support the most compact ASTs, forcing what we considered an unduly large AS. Consider the following fragment of AS for Modula-3:

```
AsDeclList := { Decl : AsDecl } .
AsDecl = AsVarDecOrList | AsTypeDecOrList | ... | AsProcDecl .
AsVarDeclOrList = AsVarDecl | AsVarDeclList .
AsVarDeclList := { AsVarDecl } .
AsVarDecl := Name : Id ; Type : [ AsType ] ; Value : AsExpr .
...
AsProcDecl
  := Name : Id ; Signature : AsSignature
     ; Block : AsBlock ; FinalName : Id .
```

An element of a Modula-3 declaration list can be either a single procedure declaration, or a sublist of constant, type, variable, or exception declarations, or revelations. In this AS, we have allowed an element of the abstract declaration list to be either a single fixed node representing any of the declaration kinds, or a list node that is a sublist of declarations, all of the same kind. As defined so far, our FS cannot insert the relevant `VAR`, `TYPE`, etc. keyword, using this AS.

Instead, we would require both an extra fixed node for each declaration sublist, with a list node beneath it, even when there was only one declaration in the sublist. The extra fixed node would then carry an FS rule to insert the `VAR`, `TYPE`, etc. These two extra nodes are quite extravagant, especially since AS nodes in Schütz are rather large.

We can support the smaller AS by generalizing the conditional formatting constructs of the FS to a CASE construct that allows more than two alternative sequences of token insertions. This construct is used in the FS rule for a declaration list. It will be applied to each element of the list and can insert one of `VAR`, `TYPE`, etc., depending on what class the list element has. The Actual FS rule is:

```
AsDeclList
  -> { CASE Decl : AsDecl
       OF MEMBER AsVarDeclOrList ( "VAR" Decl : @ 2 AsDecl )
       OF MEMBER AsTypeDeclOrList ( "TYPE" Decl : @ 2 AsDecl )
       ...
       ELSE (* PROCEDURE *) Decl : @ 2 AsDecl
       END
     ! ";"
     } .
```

Originally, we allowed a CS rule to build an AST node only for its LHS nonterminal. This made it awkward to write certain CS rules, both for LDL and for Modula-3. We extended LDL so a CS rule can specify an AST node it is to build and a different LHS nonterminal that will be used in conventional parsing.

6 Additional Work

For Schütz to work, the FS and CS must be an inverse pair. That is, if an arbitrary AST is converted to text using the FS, then reparsed using the CS, the original AST must result. At present, we rely to a great extent on the language definer's doing this correctly. Static analysis of an LDL specification makes a number of consistency checks, but the CS must still be hand written. At the time of writing, we have just begun work on automated checking of this inverse property. Ultimately, we expect this to reveal ways to infer at least large parts of the CS from the FS, thus saving the language definer considerable work.

Programmers are extremely individualistic and extremely adamant about layout. We do not expect an editor with only a single, fixed system of layout rules to be acceptable to any significant number of people. We plan to allow for multiple versions of the FS, each with its own layout rules, selected by user interface options. All versions will be mechanically checked for consistency, except for layout.

The present LDL does not specify the lexical rules of the language for the variable terminals. Spellings of the fixed terminals are, of course, implicit in a very direct way, but our current implementation does not exploit this fact. Instead, we use hand written scanners for each language. We plan eventually to add lexical specifications to LDL and to construct scanners automatically from them.

References

1. R. Bahlke and G. Snelting, “The PSG System: From Formal Language Definitions to Interactive Programming Environments”, ACM Transactions on Programming Languages and Systems, 8(4), pp. 547–576, Oct. 1986.
2. W. Barrett, R. Bates, D. Gustafson, J. Couch, “Compiler Construction Theory and Practice”, 2nd Edition, SRA, 1986.
3. R. Bagai and R. Bates , “Text Editing, Syntax-Directed Editing”, International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, L’Aquila Italy, Aug. 2001.
4. R. Bates, “Sequence-Trees: Slicing and Concatenation of Sequences in Logarithmic Time”, presented at 14th Midwest Conference on Combinatorics, Cryptography, and Computing, Oct. 2000, Wichita State University, Wichita, KS.
5. R. Bates , “Text Editor Interfaces for Semantic Editors”, SBLP2002, VI Simpósio Brasileiro do Linguagens de Programmação, Rio de Janeiro, Brazil, June 2002.
6. R. Bates , “Logarithmic-Time Slicing and Concatenation of Sequences”, Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 42, Aug. 2002.
7. M. Boshernitsan, “Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools”, Tech. Report CSD-01-1149, University of California, Berkeley, June 2001 .
8. A. Habermann and D. Notkin, “Gandalf: Software Development Environments”, IEEE Transactions on Software Engineering, SE-12(12), pp. 1117–1127, Dec. 1986.
9. H. Halme and J. Heinaenen, “GNU Emacs as a Dynamically Extensible Programming Environment”, Software-Practice and Experience, 18(10), pp. 999–1009 Oct. 1988.
10. G. Kaiser, P. Feiler, F. Jalili, J. Schlichter, “A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation”, Software Practice and Experience, 18(8), pp. 733–748, Aug. 1988.
11. W. Maddox, “Incremental Static Semantic Analysis”, Ph.D. dissertation, Report No. UCB//CSD-97-948, Computer Science Division, University of California, Berkeley, CA.
12. J. Morris and M. Schwartz, “The Design of a Language-Directed Editor for Block-Structured Languages”, ACM SIGPLAN Notices, 16(6), pp. 28–33, Jun. 1981.
13. S. Reiss, “Graphical Program Development with PECAN Program Development Systems”, ACM SIGPLAN Notices, 19(5), pp. 30–41, May 1984.
14. T. Reps, T. Teitelbaum, and A. Demers, “Incremental Context-Dependent Analysis for Language-Based Editors”, ACM Transactions on Programming Languages and Systems, 5(3), pp. 449–477, Jul. 1983

Demand-Driven Specification Partitioning

Roland T. Mittermeir and Andreas Bollin

Institut für Informatik-Systeme
Universität Klagenfurt, Austria
{roland,andi}@ifi.uni-klu.ac.at

Abstract. The paper reflects on why formal methods are quite often not used in projects that better rely on their potential. The expressive density might not be the least among them. To allow users focussed reading, the concept of specification slicing and specification chunking is introduced. An initial evaluation shows that reduction in size obtainable varies, they can be marked with larger specifications though.

1 Introduction

The crucial role of requirements elicitation and documentation is well accepted [1,2]. Some organizations even venture into requirements testing [3] to get them right. Specifying systems formally is less common though. While graphical notations like UML have reached a certain level of acceptance, specifications in languages expressing their semantics on the basis of a formal model are rare to find in industrial applications.

One has to be careful not to overgeneralize. There are certainly highly professional places around. On the other hand, there is lots of software written, where quick production of code [4] is more economical and possibly even more effective than bridging the magic leap [5] from requirements to design or from design to code by a carefully constructed formal specification. However, one gets worried when managers responsible for the production of dependable systems resort to complex explanations and make statements about compensating the disregard of formal specification by multiple (up to quintuple) modular redundancy, supported by solid reviews and heavy testing.

Certainly, many reasons can be voiced to justify avoiding formal methods. Arguments put forward are: we use N-version programming, we use only safe programming constructs, and we apply thorough testing. The fact that part of the software is written by domain experts lacking formal software-engineering education might also be among the reasons. It is rarely mentioned though.

Unfortunately, what is meant to be a valid excuse is built on shaky grounds. As safety critical systems are generally embedded systems, the reasoning of those managers (and their staff) is quite often dominated by arguments valid in classical engineering disciplines. There, triple (or higher) modular redundancy seems to be a safe bet if Poisson error rate in the individual components can be assumed. This assumption holds for most physical engineering artifacts as long as there are no common cause failures. Since faults in software are of conceptual nature, one tries to solve this issue by having the individual components developed by different teams. However, as shown in [6], N-version programming is not sufficiently effective to get rid of the effect, common education principles have engraved on software developers.

Use of "safe programming strategies" is usually a circumscription for avoiding constructs involving pointers and hence dynamic storage management. This has certainly advantages by allowing to compute upper bounds for resource consumption. However, it lowers the language level used. Had developers a chance to reason about the correctness of structures close to the application domain as well as about the mapping of these structures via clichés to the low-level constructs they are avoiding, the resulting software might be less complex and hence less buggy in the end. Without such an intermediate level, limitations in the expressive power of languages may just widen the gap to be covered by some magic leap and thus leads to further errors.

Finally, heavy emphasis on testing certainly increases quality. However, without an adequately formalized specification, formal testing is bound to be confined to structural (white-box) testing. As is well known, white-box approaches and black-box approaches are complimentary [7]. However, without a formal specification, there is no sound basis to systematically drive black-box testing or to assess black box coverage.

Remains the educational argument. While this argument might hold true, it cannot be accepted as definitive excuse. People who learned how to program should be able to learn how to write a formal specification. Hence, there must be other reasons, why they shy away from "software-engineering mathematics" while faithfully accepting the equally complex mathematics describing the physics or mechanics of their application domain.

We postulate that there are motivational factors that might be rooted in the inherent linguistic complexity of formal specifications. The next section will look more thoroughly into this argument. Based on these considerations, we propose mechanisms to support specification comprehension. With proper tool support, these concepts allow software engineers, notably maintenance personnel, to query a complex specification and obtain only the part that is relevant to the specific question at hand. The arguments are demonstrated on a set of cases analyzed.

2 The Inherent Complexity of Specifications

Let's start with an observation: people like to write code, but they do not like to read somebody else's code.

Though not based on a deep empirical survey, this statement rests on experience gained by talking to people and by observing students' as well as professionals' behavior during software maintenance. Why might this be the case? Again, based only on introspection, we postulate that it is easier to express ones owns concepts and ideas into the tight formality of a programming language than to reconstruct the concepts the original developer had in mind from the formal text written in low level code. This statement does not hold, if the original programmer adhered strictly to some standard patterns or clichés. But it holds when the code expresses a concept previously unknown to the reader. It also applies when the concept is known, but the particular form used to express it is unknown to the reader. Bennett and Rajlichs evolutionary software development model [8] postulates that the transition from the evolution phase to the servicing phase takes place when the chief-architect leaves the team. This can be quoted as macroscopic evidence for our claim.

What arguments might be raised to explain this observation? The density of linguistic expressions has certainly an impact. Humans are used to listen and talk with equal ease in their natural language. Observing a human dialog, one notices that, irrespective of the specific grammatical constructs used, it boils down to a sequence of questions and answers or assertions and counter-assertions that finally converge to the core of the message. Thus, if both parties of a conversation are reasonably sure that the respective partners' frame of reference is sufficiently adjusted such that the concept to be conveyed has actually been transferred, they might pass on to the next topic [9].

With written communication, we do not have this chance of constant probing. However, the systemic functional theory of linguistics postulates that even at such low levels as in the sentence structure, we use a head phrase (the theme) which is further explained in the rest of the sentence (the rhematic part) [10, 11]. Thus, readers of different expertise can make use of the partial redundancy between theme and rheme, respectively of the incremental nature of information transfer happening in the rhematic part.

With programming languages, this redundancy providing reassurance that some partial comprehension is still on track towards full comprehension is missing. However, at least with small programs, the well defined execution sequence among statements allows for partial comprehension and thus for incremental growth of knowledge about this piece of software. Thus, we are well capable of obtaining some understanding by performing a desk-check in the form of a program walkthrough with some assumed values. In essence, this is to inspect a dynamic slice of the program. Inspecting the program without assuming specific values bound to the program's variables is much harder.

With specifications, we have no longer a built in clue for partial comprehension. Their compactness allows the writer to succinctly pin down an idea. Due to their declarative natures, the writer does not need to worry about order of execution. As the concepts to be specified are already in the specifiers mind, the most compact way to express them is most appropriate. For the reader of a specification, the assumption that a reasonably consistent picture is already in the readers mind does not hold. The specification either needs to be grasped in its entirety or some strategy has to be identified to structure it in a way supporting partial understanding.[1]

[1] The final exams of the specification class we are teaching contains a part where students have to write a specification and a part where students have to comprehend a given specification. Although, measured in length of text, these assignments differ by a factor of 10 to 20, the time allocated to them differs only by a factor of two to three. The correctness of the results differs not at all.

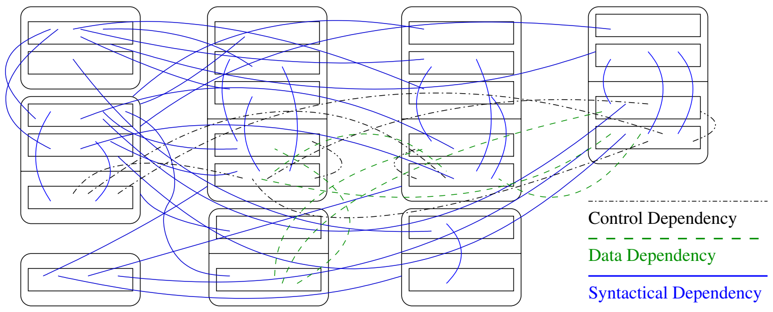


Fig. 1. Direct dependencies in the Z-specification of the birthday book [12].

Putting too much structure into a specification is usually understood to be a hint towards implementation. Hence, one has to be careful. To demonstrate our argument on a trivial example, one might consider the number of interrelationships between concepts used in the toy-specification of a Birthday-Book [12]. The BB might come close to what a minimal meaningful specification might be. The number of different dependencies between concepts (see Fig. 1) might be indicative of the fact, that it will be hard, to scale it up to an industrial size specification and still claim that the result is easy to comprehend.

Based on these considerations, we raise some arguments where partial comprehension of specifications are helpful. In the sequel, we present a concept to derive such partial specifications automatically from a full-fledged specification.

3 Demand-Driven Specification Decomposition

Usually, specifications are considered to be an intermediary product of the early phases of software development. They serve to conquer and structure the problem at hand, to harness the developer's ideas, and to reason about closure properties of requirements and the correctness of a design. They come into play again during testing. But code-level patches render them quickly outdated. For software comprehension, one usually uses tools to analyze code. Why not raising the importance of specifications by granting them life throughout the full life cycle of the software product? Keeping them up-to-date will allow using them as maintenance road-map. Pursuing this aim, one certainly has to provide tools to support partial specification comprehension, tools similar to those supporting partial code comprehension.

Focussing on the maintenance stages, a specification might serve to identify hot spots for change. While automatic tracing of change propagation [13,14] quickly reaches its limits, specifications are well suited to identify the boundaries of change propagations on the specification level and right into design, if the actual implementation is still faithful with respect to its specification. Likewise, if formal specifications are used in picking test suites, identifying the portion of the specification relevant with respect to a given changing requirement can drastically reduce the effort needed during regression testing. Pursuing these ideas, we are aiming to provide developers, notably maintenance personnel, with partial specifications that are

– substantially smaller than the full specification, and hence, easier to grasp,
– contain all relevant parts of interest,
– can be automatically derived from the full specification.

The informatics literature contains several concepts of partiality, aiming to provide an interested party just the perspective needed for a particular task. The notions of views, slices, and chunks immediately come to mind.

Views initially defined in the data-base area have been introduced to the specification literature by Daniel Jackson [15]. Data-base views are drawn from a given conceptual schema. As long as update operations via views are restricted, the view mechanism necessitates no additional integrity constraints on the schema. Jackson's specification views, though, are bottom-up constructions and the full specification of the system apparently has to allow for updates on the state space. Hence, developing software specifications via views requires additional support structure to maintain consistency of all views.

Slices have been introduced by Weiser [16] in the mid '80ies and obtained several extensions since. A slice yields the minimal part of a program that satisfies all requirements of a syntactically correct program with respect to a given slicing criterion. As such, it seems ideal for our current aim. The concept has been transferred to specifications in the work of Oda and Araki [17] and Chang and Richardson [18]. However, a direct transfer is problematic, since Weiser-slices depend on control-flow. This is not existent in specifications. Therefore, the definitions we could build upon are too liberal.

Chunks have been introduced by Burnstein [19]. Similar notions have been used in [20] for semantics preserving software restructuring. With code, a chunk comprises all those statements necessary to understand a semantically related portion of a program. The requirement of semantic completeness is weakened though. Again, with specifications the definitions need to be sharpened and recasted.

One might also mention *multi-dimensional hyperslices* [21] and the related concepts introduced in connection with aspect-oriented or subject-oriented programming. These approaches are less pertinent to our consideration though, since they are strictly generative. Our concern though is to ensure that the virtues of formal specifications can find a better use throughout the lifetime of a system.

Thus, none of these concepts of partiality developed in other sub-domains of informatics can be applied directly to specifications. However, they guide our considerations. In the sequel, the ideas leading to specification chunks and specification slices are presented. Readers interested in the formal definitions are referred to [22].

3.1 Components of Specifications

Formal specifications are expressions written in some formal language such that the primitives of the language have a formally defined semantics.

Apparently, each specification language has linguistic elements at lower granules than postulated in the previous sentence. We refer to them as *literals* of the language. Examples of literals are identifiers, linguistic operators, etc.

Minimal linguistic expressions that can be assigned meaning are called *prime objects* or *primes* for short. Primes are constructed from literals. They form the basic, semantics bearing entities of a specification. Examples are predicates or expressions. Particular specification languages will allow also for semantically richer primes. E.g. in Z, a schema or a generic type can be seen as a prime. These higher level primes are defined arrangements of literals and other primes. The important aspect is that prime objects are immutable as far as they constitute fundamental units (states and operations) specifications are built upon and that they constitute a consistent syntactic entity with defined semantics.

To support the comprehension process, independent prime objects, even if they might get complex, are usually insufficient. The peruser of a specification usually needs more than a given prime but less than the full-fledged specification to get an answer to the problem at hand. Usually, one needs a set of different primes which are to be found not necessarily in textually contiguous parts of the specification. We refer to such a set, falling short of being the full specification and falling short of any syntactical constraints observed between different primes as *specification fragment.*

Thus, the rest of the paper focusses on the following issue: given a certain (maintenance) question against a specification, what is the minimal set of primes to answer it. Following Weisers slicing idea, this fragment should have all properties of a syntactically correct specification, a property that is not fulfilled by the proposal of [18]. To solve this involves two further issues:

– What primes are needed with which kind of question?
– What is the appropriate data-structure for extracting semantically related primes and literals?

We postpone the first question, which boils down to whether one aims for slicing (i.e. wants to see the minimal portion of the specification affected by, say a changed requirement) or for chunking (i.e. looks for a minimal portion of the specification specifically related to a given part in the specification). This question becomes relevant only after a way is found to isolate an (arbitrary) part of interest and then complete this part in such a way that a minimal expression in the given specification language results that is syntactically correct and semantically complete with respect to the question at hand. The avenue towards this end is seen in the transformation of the specification into a graph and then a proper back-transformation to the original specification language.

3.2 Augmented Specification Relationship Nets

Solving the question: given some prime, which set of primes is needed to provide a maintainer with sufficient information to understand just that part of the system, requires to explore this primes neighborhood. It seems appropriate to transform the specification in a graph, analyze the dependencies in this

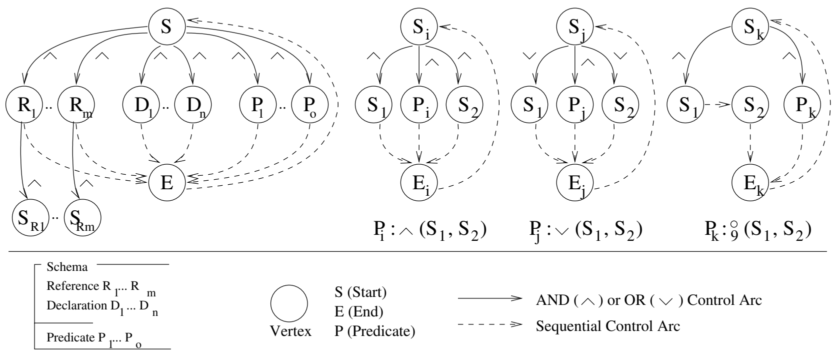


Fig. 2. Some transformations for Z-specifications into a Specification Relationship Net

graph, and perform a back-transformation of the resulting graph-fragment into the original specification language.

Using an AST as intermediate representation seems to be an obvious choice. However, the dominant structuring principle of ASTs is, according to the micro-structure of programming languages, control. This is inappropriate for declarative specifications. Good layout will help comprehending specifications, but, as with program indentation, layout has in general no semantic bearing. Getting it wrong, though, would distort readability drastically. Hence, partitioning has to retain the overall layout of the specification.On the other hand, there is a certain degree of freeness considering the placement of compound specification elements. There is no implicit canonic order comparable to the order implied by control in imperative programs.

For these reasons, we did not venture into extending ASTs to suit our purpose, but defined a graph, the *Augmented Specification Relationship Nets* or *ASRN*. An *ASRN* captures all information contained in a specification but separates the semantics contained in primes from their textual appearance in the linear or two-dimensional representation of printed text. The textual information contained in a specification is captured in an *SRN*, the *Specification Relationship Net.* There, primes are represented by arc-classified bipartite graphs such that every prime is a miniature lattice with a unique syntactic start-node and a unique syntactic end-node. Between start- and end-node are the nodes (or subgraphs) representing the elements out of which this prime is constructed. These might be literals or, with higher-order primes, references to composed primes. Depending on the relationship between the entries of the respective prime, the links from start-node to sub-prime are *AND*- or *OR*-nodes. Figure 2 schematically shows how a schema definition, consisting of several included schemata ($S_{R1}...S_{Rm}$), declarations ($D_1...D_n$), and predicates ($P_1...P_o$) would be represented in the *SRN*. It also shows *SRN*s for schema-conjunction, disjunction and composition.

The *ASRN* is an *SRN* where several semantic dependencies contained in the original specification are made explicit. These are *type* definitions, *channel* definitions (lead to the declarations of all variables in the scope of a vertex v), *value/def* definitions (definitions relevant at that vertex), and *use* arcs (leading to all vertices used at vertex v). Thus, while we assume that the specification to be transformed is correct, the *SRN* does not yet allow for static checks. Augmenting the *SRN* by the links mentioned, semantic layers are introduced that allow to identify in the *ASRN* all those vertices from which a given prime (vertex) is type-dependent, data-dependent, or simply syntactical dependent[2]. This seems to be sufficient to address the question of chunking. What is still missing, though, is a notion representing the equivalent of control dependence inherent in code-slicing.

To address this issue, one has first to reflect on what constitutes the equivalence of control in specifications. Apparently, preconditions of operations define, whether an operation will be executed. How the precondition is represented and how it will be evaluated in the final program is immaterial at this point. But for the specification, it suffices to observe that a situation not satisfying a precondition implies that the transformation (or action) expressed by this operation should never be executed while a situation satisfying the precondition is a situation where the operation might be executed, i.e., it potentially obtains control. In certain specification languages, e.g. *VDM*, pre- and post-conditions are explicitly defined. In others, like *Z*, they have to be computed. As the algorithm must not depend on the specifiers adherence to style conventions, we opted for the heuristic that predicates involving no state changes or external activities are considered pre-conditions. Those involving state changes (i.e., contain decorated variables) are considered post-conditions.

3.3 Chunks and Slices

With these preparatory remarks, chunks and slices can be defined. Both, chunks and slices, are abstractions of the given full specification defined "around" a specific point of interest. In program slicing, this is referred to as slicing criterion. It refers to a variable in a given statement (at position). Here, we look at a variable (literal, l) within a given prime (vertex v_l or *articulation prime*). The maintainer might be interested in the interaction of l with some other variables of the specification. Usually, those will be other variables referred to in the same prime, but those can be any variables appearing anywhere in the specification. One might summarize them under the term "potentially interesting targets". They can be listed in an additional argument Θ in the *abstraction criterion* (*slicing-, resp. chunking-criterion*).

Definition 1. *An* a*bstraction criterion for a specification is a triple* (l, Θ, Γ)*, where* l *is a literal in a prime of the specification (represented in the ASRN by*

[2] This encompasses dependencies from vertices needed for the proper definition of the prime under consideration. Thus, it covers type-dependencies as well as channel-dependencies.

vertex $v_l \in V$). Θ is a subset of the set of symbols (variables) defined or used in the ASRN, and Γ is a set of dependencies, $\Gamma \subseteq \{C, D, S\}$, with C denoting control dependency, D denoting data dependency, and S denoting syntactical dependency.

If in the above definition the literals and primes listed in Θ all occur in the prime, l is taken from and Γ is set to a value different from the full set $\{C, D, S\}$ we refer to it as chunking criterion, one obtains a conceptual (hyper-)plane along the dependency categories listed in Γ, originating at the articulation-prime and transitively extending over the full specification. This hyper-plane is referred to as *static specification chunk* or *SChunk*. Compared with the notion of Burnstein-Chunks, where locality was aimed at, this provides for full projection.

The following definition captures a chunking concept similar to Burnstein's ideas on the specification level. It, the *BChunk*, is made up of the intermediate context of a prime according to some filter criterion.

Definition 2. *A static BChunk referred to as BChunk(PL, Θ, Γ) of S on chunking criterion (PL, Θ, Γ) (where $v_p \in V$ and $V_P \subseteq V$ and $\Gamma \neq \varnothing$) is a subset of vertices $V_{PL} \subseteq V$ such that for all $v \in V$ holds: vertex $v \in V_{PL}$ iff v is either directly data-dependent with respect to the variables defined in Θ on V_P and $D \in \Gamma$, or directly control-dependent on V_P and $C \in \Gamma$, or syntactical dependent on V_P and $S \in \Gamma$.*

The definition of BChunks just focusses on which other vertices than the criterion need to be included in V_l. This suffices, since the selected vertices represent primes in the original specification and these primes are related by the *SRN* covering the whole specification. The external representation of the specification chunk is the result of a back-transformation of all those vertices (primes) in the *SRN* that belong to V_l. The *SRN*-arcs will assure that the result will be a particularly focussed specification fragment.

As the above definition for chunks was chosen to transitively cover the full specification according to the respective chunking criterion in a particular (set of) dimension(s) of interest, it can be directly extended to the definition of a specification slice. As with program slices, we require that a specification slice is a syntactically correct specification which is, with respect to the slicing criterion, also semantically complete. This will be obtained, if all kinds of dependencies are included in the specification criterion, i.e., $\Gamma = \{C, D, S\}$:

Definition 3. *The* static specification slice *SSlice(l, Θ) on a slicing criterion (l, Θ) corresponds to the full static specification chunk SChunk$(l, \Theta, \{D, C, S\})$.*

As slices adhere rigidly to the constraint of syntactical correctness, the resulting specification stays syntactically correct and remains (with respect to the specification criterion) understandable. Nevertheless, as all dependency types are included in the slice, the resulting slice is normally bigger than the chunks derivable. With tightly interwoven specifications, the slice might get as big as the original specification.

4 Case Studies

Based on the definitions mentioned above, a toolkit for slicing and chunking Z-specifications has been developed. We applied the partitioning algorithms on several specifications, among them are the Birthday-Book [12], the Elevator [12, 18] and the Petrol-Station, a specification we used as course assignment.

None of these specifications comes close to industrial size. They just served as test- and demonstration cases during method- and tool development. Nevertheless, they can be considered as proof of concept for the arguments raised and to show that the concepts put forward do reduce complexity of size and, therefore, the overall complexity of the specification comprehension task, if full understanding is not really needed.

The literature in the field of software metrics is vast. The various measures of size are at least correlated. The proposals to measure inherent complexity are quite different though. However, there is agreement that size is a prominent scaling factor even for other forms of complexity [23].

Measuring size of a specification by number of characters, number of lines (as analog to *LOC*) or number of predicates (as analog to *KDSI*) seems not very attractive. Such measures miss the crucial point of compactness of formal specifications referred to in the introduction. Hence, we decided to measure the complexity of a specification by the number of concepts and the number of interrelationships between concepts needed to formally trace it in order to obtain a meaningful specification fragment. Chunks or slices would qualify for such meaningful fragments. Hence, the number of vertices and arcs in an *ASRN* seems to be a fair candidate metric.

However, one should not take their raw value, since many arcs in the *ASRN* are just needed to capture layout information and external structure of the specification. Their use is not burdening the comprehender. The links that implicitly relate concepts contained in the specification, be they implicit data- or control-dependencies, or be they syntactical dependencies that get, when interpreted, a particular semantic meaning, cause interpretative load.

Table 1 shows the number of vertices / number of nodes in the *ASRN* for the three specifications mentioned, and also quotes the number of control-, data- and syntactical dependencies.

In the *ASRN* representation, even the simple BB specification consists of 84 vertices and 302 arcs. Since the BB is about the simplest non-trivial specification one might think about, these numbers seem already high. But one must not be frightened. Only 142 arcs have semantic significance. The remaining 160 capture layout information needed for the back-transformation. Thus, only 142, i.e., 128 for syntactical dependencies, 7 for control and 7 for data-dependence carry intellectual load. Due to this different significance of link-weights, the reduction-percentage is of limited significance. Readers might get a better appreciation of the benefit of this approach when comparing the overall reduction in the various link categories tabulated in the second line of each experiment.

When applying a static Slice $SSlice(n, birthday)$ (where l represents the prime object "$birthday' = birthday \cup \{name? \mapsto date?\}$" in the Add-schema), the

Table 1. Complexity of the *ASRN* representation of three different specifications. The rows represent the number of nodes and arcs after applying slicing and chunking functions on the specification.

Original Spec.	BirthdayBook	Elevator	Petrol-Station
ASRN Metric	84V / 302A 128s - 7c - 7d	344V / 2835A 634s - 386c - 1160d	134V / 668A 251s - 36c - 120d
$SSlice(l, \theta_1)$	58V / 201A 87s - 5s - 4d	240V / 2105A 464s - 260c - 881d	109V / 542A 211s - 26c - 92d
$SSlice(l, \theta_2)$	53V / 184A 79s - 4c - 4d	175V / 1407A 330s - 77c - 565d	103V / 503A 198s - 16c- 82d
$SChunk(l, \theta_2, \{S, C\})$	38V / 130A 57s - 2c - 2d	35V / 146A 52s - 5c - 9d	39V / 182A 71s - 5c - 26d
$SChunk(l, \theta_2, \{S, D\})$	46V / 158A 62s - 4c - 2d	113V / 698A 174s - 63c - 90d	86V / 403A 154s - 16c - 43d

V ... vertices (including syntactical comment node vertices) - A ... arcs in the *ASRN*
l ... vertex in the *ASRN* representing a prime object in the specification
θ_1, θ_2 ... slicing with different set of literals (see text for explanation)
s, c, d ... number of arcs representing syntactical, control and data dependencies.

overall complexity of the net (and the specification) is distinctly reduced. (More important is the reduction of the overall number of arcs representing syntactical-, control- and data-dependency). The second slice on birthday is also on this predicate in the Add-schema, but $\Theta 2$ consists only of $\{birthday\}$ in contrast to $\Theta 1 = \{name, birthday\}$ in the first slice. The same holds for the application of two chunking functions onto the same abstraction criterion. When transforming the sliced *ASRN* back to Z, the remaining specification only consists of the BB state-schema, the InitBB schema, the Add- and the Remove schema. The schemata Find and Success are pruned from the specification and the remaining specification has been reduced by about 30 percent. The difference in Θ has with this example only a minor effect.

The difference between $\Theta 1$ and $\Theta 2$ is significant though in the much larger Elevator specification[3].

With the full panel of variables, reduction is not impressive. Slicing for a single variable yields a reduction of 50 % though. The slices have been computed with the prime $Requests' = Requests$ in the FloorButtonEvent-Schema. If, for this schema, only syntactical semantic and control dependencies are considered, the resulting chunk has only 35 (instead of 344) vertices and 146 (instead of 2835) links. The Petrol Station assumes a medium position in this table.

Reductions of even higher magnitude can been found when looking at the numbers of dependencies in the *ASRN*. When looking at the first chunk of the elevator example, one will find out that 376 control dependencies and 1160 data de-

[3] In our example we used $\Theta 1 = \{Requests, CurrentFloor, Door, UpCalls, Down\text{-}Calls\}$ and $\Theta 2 = \{Requests\}$ for the generation of the slices.

pendencies are reduced to just 5 control-dependencies and 9 data-dependencies. In that case this is a reduction of at least 98% – an impressive number of dependencies whose elimination is tremendously reducing the complexity of the remaining specification.

Apparently, the reduction factor depends on various aspects. The compactness of a specification and the interrelationship between state variables not being the least among them. Nevertheless, our preliminary analysis suggests that the effect obtainable by slicing and chunking rises with the size of specifications under consideration. Thus, our claim that these concepts are meant for "industrial size" specifications seems substantiated.

5 Summary-Conclusion

The paper introduced a mechanism for automatically deriving semantically meaningful fragments of formal specifications. The work was motivated by helping software developers to comprehend large specifications in situations where their current interest is confined to a particular aspect only. Nevertheless, these developers have to be sure that no relevant portion of the specification which directly or indirectly influences the part they are focussing on, will be ignored in their partial comprehension.

Chunks and slices have been discussed as prototypical abstractions that proved their value already in program understanding. Initial assessments are encouraging, since the effects of focussing attention tends to increase with larger specifications. With large specifications, the $ASRN$ can no longer be explicitly displayed. However, it can still be computed efficiently, since the approach builds only on proven compiler technology.

Departing from the archetypical specification fragments of chunks and slices, we plan to further refine the approach presented in this paper, such that more specific issues arising during software maintenance can be answered by even narrower focussed specification fragments containing all the information needed for the task at hand.

References

1. Boehm, B.W.: Software Engineering Economics. Prentice Hall, Englewood Clifss, NJ (1981)
2. Kotonya, G., Sommerville, I.: Requirements Engineering. 2nd edn. John Wiley & Sons, Ltd. (1998)
3. Graham, D.: Requirements and Testing: Seven Missing-Link Myth. IEEE Software **19** (2002) 15–17
4. Boehm, B.: Get Ready for Agile Methods, with Care. IEEE Computer **35** (2002) 64–69
5. Potter, B., Sinclair, J., Till, D.: An Introduction to Formal Specification and Z. Prentice-Hall Intl. (1991)
6. Knight, J.C., Leveson, N.G.: An Experimental Evaluation of the Assumption of Indepencence in Multiversion Programming. IEEE Trans. on Software Engineering **SE-12** (1986)

7. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software Systems. John Wiley & Sons, Inc. (1995)
8. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: A roadmap. In Finkelstein, A., ed.: The Future of Software Engineering 2000. ACM press (2000) 73–87
9. Stewart, C.J., Cash, W.B.: Interviewing: Principles and Practices. 2nd edn. Wm. C. Brown, Iowa (1978)
10. Daneš, F.: Functional sentence perspective and the organization of the text. In Danes, F., ed.: Papers on Functional Sentence Perspective, Academia, Publishing House of The Czechoslovak Academy of Sciences, Prague (1970) 106–128
11. Halliday, M.: An Introduction to Functional Grammar. Edward Arnold (1985)
12. Spivey, J.: The Z Notation: A Reference Manual. Second edn. Prentice Hall International (1992)
13. Pirker, H.: Specification Based Software Maintenance: A Motivation for Service Channels. PhD thesis, Universität Klagenfurt (2001)
14. Pirker, H., Mittermeir, R.T.: Internal service channels: Principles and limits. In: Proceedings International Workshop on the Principles of Software Evolution (IWPSE'98), IEEE-CS Press (1998) 63–67
15. Jackson, D.: Structuring Z Specifications with Views. ACM Trans. on Software Engineering and Methodology **4** (1995)
16. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, IEEE (1982) 439–449
17. Oda, T., Araki, K.: Specification slicing in a formal methods software development. In: 17th Annual International Computer Software and Applications Conference. IEEE Computer Socienty Press (1993) 313–319
18. Chang, J., Richardson, D.J.: Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California (1994)
19. Burnstein, I., Roberson, K., Saner, F., Mirza, A., Tubaishat, A.: A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In: TAI-97, 9th International Conference on Tools with Artificial Intelligence, IEEE press (1997)
20. Mittermeir, R., Rauner-Reithmayer, D.: Applying concepts of soft-computing to software re(verse)-engineering. In: Migration Strategies for Legacy Systems. TUV-1841-97-06 (1997)
21. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proc. 22nd Internat. Conference on Software Engineering, ACM and IEEE press (1999) 107–119
22. Bollin, A., Mittermeir, R.T.: Specification Fragments with Defined Semantics to Support SW-Evolution. In: ACCIT/IEEE Proc. of the Arab-International Conference on Computer Systems and Applications (AICCSA'03). (2003)
23. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigourous & Practical Approach. Second edn. International Thompson Publishing Company (1996)

Source-to-Source Transformation in the Large

Thomas Genssler and Volker Kuttruff

Forschungszentrum Informatik Karlsruhe
Haid-und-Neu-Strasse 10–14
76131 Karlsruhe, Germany
{gensslex,kuttruff}@fzi.de

Abstract. In this paper we present Inject/J, both a language and a tool for specifying complex source-to-source transformations of Java programs. The focus of Inject/J is on "transformation in the large" that is, modification of large object-oriented software on the design level. We first introduce the meta-model of our transformation language. This meta-model provides a conceptual view on object-oriented software by capturing relevant design entities. It also defines a number of conceptual analysis and transformation operations together with their code-level semantics. The entities of the meta-model, together with the respective operations, constitute the primitives of our transformation language. We discuss the main features of this transformation language and illustrate how it can be used to perform complex transformation tasks.

Keywords: Reflective programming, Software development environments

1 Introduction

There is one constant in the life of software: it changes. It may change because of changing requirements or due to bug-fixing. Changing large-scale software is a difficult task that calls for tool support, especially if the changes have global effects. Automated source-to-source transformation is one possible solution. With source-to-source transformation we denote the purposeful modification of a piece of software by changing the source code directly. The most prominent example is probably refactoring [16,10]. In this paper we present an approach to the specification of complex source-to-source transformations and their application to large object-oriented software. We focus on design level transformations. Design level transformations modify entities (classes, methods etc.), which contribute to the design of a system, as well as the interactions among these entities at runtime (control flow along method invocations or accesses to attributes of a class).

When transforming large software, one of the main problems is the huge amount of information. Therefore, we need *abstraction* and *locality.* We want to be able to focus on only those parts of the system we want to transform (certain classes or specific set of method calls) rather than having to understand the system in its entirety. However, sometimes these parts do not comply to the decomposition mechanism used in the programming language (i.e. classes or methods). Consider the case that you want to replace direct attribute accesses with calls to the respective set- and get-methods. The set of accesses probably crosscuts a large number of classes. Thus we need a technique to describe, identify, and transform these possibly distributed elements in a coherent way.

Another difficulty in automated software transformation is *correctness*. We need a notion of correctness, even when allowing general-purpose transformations. The transformation result must at least be correct according the syntax and the static semantics of the programming language. In addition, we need a mechanism to impose stricter constraints, e.g. to ensure behavior preservation.

Last but not least, we need *extensibility*. Since there is no complete list of all possible transformations, we need to be able to specify new transformations in terms of existing ones.

Our approach is based on three pillars: a meta-model, a language and a tool. We first define a two-layer meta-model for large-scale source-to-source transformations. The upper layer introduces a conceptual view on object-oriented languages, which consists of common entities of these languages and a rich set of conceptual analysis and transformation operations. The lower layer defines the mapping between the conceptual view and the actual language semantics.

We then define a transformation language based on our meta-model. This language provides a powerful and expressive means to specify complex transformation operations. It provides language constructs to query the model, to match patterns, and to perform transformations.

Finally, we provide a tool that constructs the conceptual model from source code, performs transformations specified in our transformation language and regenerates the new code.

The remainder of this paper is organized as follows. In Sect. 2, we introduce our conceptual meta-model and show how it maps to the Java programming language. Section 3 is devoted to our transformation language. We show how our approach can be used to specify and perform practical source-to-source transformations on object-oriented code. We also sketch our tool Inject/J. After a discussion of our approach in Sect. 4, we give an overview on related work in Sect. 5 and conclude in Sect. 6.

2 A Meta-model for Source-to-Source Transformation in the Large

In this section we introduce our meta-model and show how it maps to Java. Our meta-model (see Fig. 1) consists of two layers. The upper layer defines a conceptual view for design level transformations. Its goal is to provide an abstraction layer for the specification and application of transformations with non-local effects while shielding the user from the complex syntax tree of the underlying language. The lower layer defines a language-specific mapping of the conceptual entities and design level transformations.

Typical design level transformations with non-local effects are, among others, method and attribute refactorings (e.g. encapsulate field), class-level refactorings (e.g. replace inheritance with delegation), or modifications of the control- and

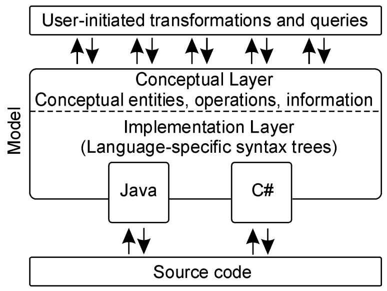

User-initiated transformations and queries
Conceptual Layer
Conceptual entities, operations, information
Implementation Layer
(Language-specific syntax trees)
Model
Java
C#
Source code

Fig. 1. Inject/J Meta-Model

data-flow among objects (e.g. replace method with method object or code instrumentation along delegation paths). Hence this model captures the following common elements of object-oriented languages that either contribute to the design of a program or represent significant points in the interaction among design entities:

– *Declaration of structural entities*: Package, Class, Parameterized Class, Array, Type Parameter, Attribute, Local Variable, Formal Parameter
– *Declaration of behavioral entities*: Method, Constructor, Property, Exception handler
– *Entities modelling control or data flow*: Accesses to declared elements, Access paths, Return, Exception, Assignment

The different types (class, parameterized class, array, type parameter) are used to reflect a language's type system. For our Java implementation, classes, interfaces, inner classes as well as base types are all mapped to the model entity 'class'. Since Java does not (yet) support type genericity, parameterized classes and type parameters are not used in the Java mapping. Attributes, local variables and formal parameters map in a straightforward way.

Behavioral entities define elements that take part in the system's control flow. The behavioral entities methods and constructors as found in Java map to the corresponding model entities. Java's class and instance initializer are mapped to the 'constructor' model entity. Properties, as found in other object-oriented languages like C#, do not exist in Java and are therefore not considered.

In our meta-model, accesses are references to structural or behavioral entities, e.g. method invocations changing the control flow or attribute/variable accesses describing the data flow. They map directly to the corresponding Java language constructs. The 'return' model element, as well as the 'exception' and the 'assignment' entity map to Java in a similar way.

2.1 Information Provided by the Model

In general, each model entity has a set of attributes providing detailed information. Some of them can be gathered directly from a system's syntax tree, like the name of a structural entity, others need an explicit semantic analysis or more advanced computation.

The information provided by model element attributes can be categorized as follows:

– syntactical information, e.g. class, method and attribute name
– basic semantical information, e.g. static type information, functions to test inheritability of class members
– crossreference information
– basic information for metrics calculation, e.g. lines of code for a class/ method, number of nodes/edges in the control flow graph of a method

Exact computation of this information, as needed in our model (e.g. for precondition checking), requires a closed world. Additionally, there is also no support for reflective programs (e.g. programs using Java reflection).

2.2 Conceptual Transformations

Conceptual transformations are the basic operations to modify a model instance. However, a single conceptual transformation can result in an extensive reorganization of the source code of the system. Changing a method's signature is an example of such a conceptual transformation, since all call sites of the method must be adapted as well. In general, a conceptual transformation can be characterized by the fact that in addition to the entity to be changed (*primary transformation*), all structures depending upon this entity are also adapted automatically (*secondary transformation*). This basically means that a conceptual transformation can be used with only local knowledge about its effects. An overview of our conceptual transformations is given in Fig. 2.

In our model, each conceptual transformation is a quadruple $(s, pre, t, post)$ with:

1. *Signature s*: The signature of a conceptual transformation is part of the upper model layer. It defines the name and the parameters of the transformation.
2. *Set of pre-conditions pre*: Each conceptual transformation has a set of preconditions, which ensure the correctness of the transformation result, according to the syntax and the static semantics of the underlying programming language. Since these pre-conditions are language-specific, *pre* is part of the language mapping
3. *Code-level Transformations t*: Code-level Transformations are defined in terms of the necessary primary and secondary transformations, that is, all language-specific source code modifications needed to perform: a) the actual primary transformation and b) all necessary secondary transformations to make sure that the result remains compilable. Some transformations may require user interaction (e.g. the default value in a method access if adding a new parameter to the method declaration).
4. *Set of post-conditions post*: Post-conditions define properties that the code fulfils after the transformation, given that the transformation succeeded.

	Package	Class	Method	Constructor	Attribute	Property	Local Variable	Class Access	Method Access Constructor Access Attribute Access Access Local Var.	Access Path	Assignment	Return	Exception	Exception Handler	Remarks
create	×	×	×	×	×	×									
delete	×	×	×	×	×	×	×								
change signature															
modifier		×	×	×	×	×									
name	×	×	×		×	×	×								
type			×		×	×	×								
superclasses		×													
parameter			×	×											
exceptions			×	×											
	The transformations below allow the insertion of arbitrary code fragments														
replace body			×	×										×	
replace ...								×	×	×	×	×	×		
before							×		×	×	×	×	×		before control flow (=cf) reaches entity
after							×		×	×	×				after cf passed entity
before entry			×	×											before cf passes to entity, parameters are already evaluated. Optional 'from' restricts places of invocation
before return			×	×										×	before cf leaves entity after normal completion
before failure			×	×										×	before cf leaves entity after abnormal completion. Optional 'with' restricts failure reasons
before exit			×	×										×	before cf leaves entity (normal or abnormal completion)
after entry			×	×										×	after cf passed to entity
after return			×	×											after cf returned to place of invocation (restricted by optional 'to') after normal completion.
after failure			×	×											after cf returned to place of invocation (restricted by optional 'to') after abnormal completion
after exit			×	×											after cf returned to place of invocation (restricted by optional 'to')

Fig. 2. Conceptual transformations supported by the transformation model

All our conceptual transformations have been specified in this manner. Some conceptual transformations, like `before entry`, allow the insertion of arbitrary code fragments. This insertion is also guarded by a set of pre-conditions, since the code fragments must match their new context. For example, pre-conditions ensure that code fragments are syntactically correct, that they do not create dead code and that they do not redefine used variables. While inserting new code fragments, it is often necessary to adapt the context. For example, if adding a code fragment before an access to an attribute `a`, a method access `m(x,a)` using this attribute must be transformed as well. This is due to the fact that the new code fragment cannot be inserted inside the parameter list. If necessary, such context adaption is performed automatically by the conceptual transformation. For each context adaption, the conceptual transformation guarantees an unchanged evaluation order, e.g. by using temporary local variables.

For each conceptual transformation, detailed Java-specific pre- and post-conditions have been formalized. These conditions are part of the transformation model and ensure that the transformed source code remains compilable.

3 The Inject/J Software Transformation Language

In the previous section we have sketched our meta-model, as well as a model mapping to Java. This meta-model provides us with the necessary functionality to reason about software and to transform it on the design level. We are now ready to introduce our software transformation language Inject/J. Inject/J is a mixture of a pattern-based transformation system (i.e. so-called detection patterns are used to identify structures which are to be transformed), and a programmatic one (i.e. the actual transformations are described imperatively). The main goal of the language is to make specifying transformations as simple as possible for a software engineer.

3.1 Language Basics

The Inject/J software transformation language is a dynamically typed scripting language that serves to specify complex transformation operators in terms of model entities and conceptual operations. It provides full access to the upper meta-model layer. It has language constructs to navigate through a model instance, to select model entities according to their properties and to perform model transformations.

Our language provides a number of built-in types. These types comprise all entities defined by our model (e.g. class, method, access), as well as a number of general-purpose data types (e.g. strings and lists). The language has quantors (i.e. *foreach* and *exists*) and provides the concept of code blocks [1].

To declaratively describe code patterns that are to be transformed, the concept of *detection patterns* is provided. Detection patterns serve to detect these code patterns in the system in order to perform the necessary transformations. Detection patterns are basically graph patterns that match – according to certain conditions – sub-graphs in the graph representing the software system. A detection pattern consists of a signature and a body. The signature contains an optional list of parameters to configure the pattern and a mandatory selector. The selector serves to distinguish different matches. The body of a detection pattern consists of an optional initializer, an optional list of pattern variables, an optional list of additional functions and a mandatory list of conditions. The list of conditions specifies the properties an instance of a particular pattern needs to have. The following code shows how one could express our introductory example using a detection pattern. To instantiate the pattern, we use two pre-defined sets (*classes* and *methods*) that contain all classes or methods of the system.

[1] Code blocks have the form `'['` `<optParams>` `'|'` `<exprList>` `']'`. They can be bound to variables and passed as parameters.

```
script Logger {
  m = classes.filter('mypackage.MyClass').filter('int myMethod()');
  m.beforeEntry('srcPackage.*', ${Logger.getInstance().print("Calling <m.signature>");}$);
}
```

```
Java source before transformation     after transformation
...                                   ...
o.f(anotherMethod(), mc.myMethod())   <type> tmp1 = anotherMethod();
...                                   // begin injected code
                                      Logger.getInstance().print("Calling int myMethod()");
                                      // end injected code
                                      int tmp2 = mc.myMethod();
                                      o.f(tmp1, tmp2);
                                      ...
```

Fig. 3. Inject/J script and its code level effects

```
pattern MyPattern():(c, m) {
 conditions { (c.package == 'mypckg'), (m in c.methods); }
}
...
foreach p in MyPattern()(classes, methods) do { ... }
```

The selector of the used pattern consists of a class and a method. The elements of the selector are bound to the elements of the cross-product of sets used in the pattern instantiation (e.g. *classes* and *methods*). Two structures matched by this pattern are different from each other, iff the binding of at least one of the parameters of the selector is different, i.e. `('MyClass','myMethod')` differs from `('MyClass','myOtherMethod')`. We call structures that match a particular detection pattern *pattern instances*. The order in which pattern instances are processed is not defined.

After identifying pattern instances (or navigating to a certain location in the model instance), we can apply transformations. The Inject/J language provides the same conceptual transformations as defined in our model (see Fig. 2 for a summary). Conceptual transformations can easily be composed to construct more complex transformations. Each conceptual Inject/J transformation automatically takes care of language-specific code-level effects (such as flattening of expressions). Each conceptual transformation guarantees that the code remains compilable. It also guarantees that the execution or evaluation order will not be changed (i.e. in case expressions have to be split, the evaluation order is guaranteed to remain the same). Figure 3 gives an example of a transformation of distributed elements (i.e. method call-sites) that potentially requires a large number of secondary transformations. The example also illustrates code-level effects of the transformation.

Matching detection patterns (basically graph patterns) and transforming them (basically graph rewriting) may lead to *non-termination*. This is the case, when a transformation applied to a pattern instance constantly creates new pattern instances. To avoid this, we have chosen the following approach: the set of all pattern instances (of possibly different patterns used in one script) is constructed, before any transformation the pattern is possibly involved in, takes

```
pattern CWMR( methodThreshhold, similarityPercentage, clusterSize) : (c) {
  vars clusters;
  init {
    threshold = c.attributes().size() * similarityPercentage;
    tmpMeths = c.methods().clone();
    cluster = []; clusters = [];
    // Find similar methods. Similarity is defined by the amount of common attributes uses
    foreach m_i in c.methods do {
      tmpMeths.removeElement(m_i);
      cluster.addElement(m_i);
      foreach m_j in tmpMeths do {
        tmpAtts = m_i.accessedAttributes().intersect(
          m_j.accessedAttributes().intersect(c.attributes()) );
        // attributes used by both methods
        if( tmpAtts.size() > threshold )
          cluster.addElement(m_j);
      }
      if(cluster.size() > clusterSize)
        clusters.addElement(cluster.clone());
      cluster.clear();
    }
  }
  conditions { (c.WMC() > methodThreshold), (clusters.size() >= 2); }
  public getClusters() { return clusters; }
}
```

Fig. 4. Detection pattern for 'Classes with multiple responsibilities'

place. New pattern instances that are created during the transformation process are not considered.

The case that two pattern instances overlap each other also has to be considered. The transformation of one of these instances could "destroy" another one, i.e. change its properties in a way that it is no longer a pattern instance. To deal with this, each pattern instance is analyzed before it is processed, whether parts of it have been already modified. If so, the pattern conditions are re-checked for this instance. If they still hold, the pattern instance can be transformed. Otherwise, the instance is ignored.

3.2 Example

In this section we show, how Inject/J can be used to fix a common design problem, e.g. classes with different responsibilities. Each class that implements more than one key abstraction (responsibility) is considered a problem (e.g. in [17]). We show how the description of the problem can be formalized using detection patterns. We also specify the necessary transformations to correct the problem.

Classes with different responsibilities are usually complex and their attributes and their methods can be clustered in disjoint, highly cohesive sets with pair-wise low cohesion. To detect such classes, complexity metrics such as Weighted Method Count (WMC) [7] and cohesion metrics, such as Tight Class Cohesion (TCC) [5] can be used. [10] proposes to use the refactoring "Extract class" to split classes with multiple responsibilities.

```
 1 use library cwmr;
 2 script FixCWMR {
 3   foreach p in CWMR(...)(classes) do {
 4     foreach cluster in p.clusters do {
 5       name = "";
 6       ask("Qualified name of the new class:", name);
 7       clusterAtts = cwmr.getAtributesForCluster(cluster);
 8       extractClass(p.c, cluster, clusterAtts, name);
 9     } // foreach cluster
10   } //foreach p
11 } // end of script
```

Fig. 5. Transformations for 'Classes with multiple responsibilities'

The detection pattern we use to detect instances of our problem is shown in Fig. 4. It detects complex classes with cohesive clusters of methods according to usage of class variables. For reasons of clarity we use a rather naive approach to formalize our problem. We refer the reader to work on coupling and cohesion metrics for an extensive discussion on how to formalize this problem.

Note that this detection pattern identifies only problem *candidates*. This is due to the fact that the problem is described in a fuzzy way (using percentages, thresholds etc.). We need human interaction to decide, whether a problem candidate is an actual problem structure. Our language provides support for that. The respective code has, however, been omitted from Fig. 5.

Once having confirmed that the candidate is a real problem instance, we can apply the necessary transformations. In Fig. 5, we only show the use of the complex refactoring `Extract Class` for reasons of clarity (line 8).

3.3 The Inject/J Tool

Together with our transformation language, we developed the tool Inject/J [12] to apply transformations to a software system. Inject/J is based on the meta-programming library Recoder for Java ([1] [15]). Recoder provides the necessary infrastructure to build an abstract syntax tree together with semantic information (e.g. cross-reference and type information). Recoder also allows to modify this syntax tree and to re-generate code from it. Recoder takes care of layout preservation of source files (i.e. indentation and comments are maintained). In order to compute cross-references, Recoder also analyzes Java byte-code files. However, modification is only allowed for software available in source code. Inject/J interfaces with the Recoder syntax tree and constructs its own model. Inject/J scripts have full access to this model that is, they can access model information and perform model transformations.

4 Discussion and Open Issues

The main goal of our approach was to provide a software engineer with a simple yet practical tool for source-to-source transformation of large object-oriented systems. To be able to deal with large systems, we provide a conceptual model that shields the user from the complexity of the underlying language model and provides significant abstraction. For structure-rich Java systems, our approach usually reduces the number of model elements – compared to the complete syntax tree – by about 60%[2]. For algorithmic-centric systems, the reduction is even better. Besides this information reduction, our model together with the Inject/J language allow for the specification of complex transformations with local knowledge about the system. The software engineer can focus on those entities he actually wants to transform, while all necessary secondary transformations are taken care of automatically by our tool. With detection patterns one can describe, match and transform structures in the code that are potentially distributed throughout the entire system, in a coherent way. Inject/J also supports pre- and post-conditions to impose stronger constraints than those defined as part of the conceptual transformations. For this purpose, the language provides full access to analysis information provided by the transformation model. We have specified behavior-preserving pre-conditions for 26 design-level refactorings presented in [10].

Besides the correction of design problems, our approach can also be used for different purposes. One example is code instrumentation, possibly as a prestep of compilation, another is code generation (e.g. in the context of Java EJB systems). We have also been working on the integration of techniques from Adaptive Programming (AP) [14]. In [20] we show how our approach can be extended to support introduction of new functionality along inheritance and delegations paths. Another area is the automated application of design patterns [19] [8], as well as the modular specification of design patterns as presented in [11]. In the latter case, the reusable parts of a design pattern implementation can be specified using an Inject/J script.

There are, however, some open issues. Our approach does not consider reflective features of programming languages, such as Java reflection. This is an intrinsic limitation of tool-supported software transformation. A possible solution is to use heuristics or conservative approximations to deal with this problem. Another intrinsic property of software transformation is that there is no support for incomplete code (i.e. code that is not compilable). Currently, we also do not support the automatic inference of pre-conditions for composite transformations. This means that a long-running transformation may fail "in the middle" and must thus be rolled back completely. A possible solution to this problem is described in [18]. Another problem is that design entities may not only be referenced from other design entities but also from different artifacts such as Javadoc comments or design documents. Currently we do not support secondary transformations to keep artifacts other than the code itself consistent (e.g. by renaming a Javadoc reference to a class when the class is renamed).

5 Related Work

Several approaches have been proposed in literature to support software transformation in the context of object-oriented software evolution – the most visible

[2] e.g. Recoder for Java [1], 85kLOC, 500 classes; reduction: ~65%

among them is without doubt refactoring [16] [10]. Although refactorings define a set of conceptual transformations together with pre- and post-conditions, their code-level effects as well as the needed analysis information is only semi-formalized if at all. There is no notion of extensibility (apart from sequential execution of several, otherwise independent refactorings). Pattern detection support, i.e. identifying structures where to actually apply refactorings, other than the rather informal "bad smells" in [4] and [10], is not provided. There exist a number of efforts to improve refactorings. [21] proposes a language-independent model for refactoring. This model provides conceptual analysis and transformation operations together with a formalization of their language-specific code-level effects. [18] proposes a technique to derive pre- and post-conditions of complex refactorings from the conditions of their constituent refactorings. However, neither extensibility nor pattern detection support are considered. There are a number of general-purpose transformation systems that address source-to-source transformation. Most of them are so-called pattern-based transformation systems, e.g. Design Maintenance System (DMS) [9], ASF+SDF [2], JATS [6], to name a few. A source pattern, described in a special pattern language, is searched in the program representation and replaced by a target pattern. Both, the source pattern and the target pattern are declaratively specified in so-called rewrite rules. Due to their declarative nature, pattern-based transformations theoretically offer an expressive, simple and easy to understand means to specify complex transformations. In practice however, with pattern languages it often gets very inconvenient to specify transformations which rely on a lot of contextual information. Our detection patterns compare to source patterns in rewrite rules, although we use a programmatic approach to specify the actual transformations.

Detection patterns also compare to join-point descriptions, used for aspect composition in aspect-oriented programming ([13]). In aspect languages (e.g. AspectJ [3]), one can specify so-called point-cuts – sets of points in a program (method entry/exit, method call, object access etc.) where aspect code is woven. Aspect languages, however, do not support general design transformations (e.g. there is no support for behavior-preserving restructuring).

6 Summary

In this paper we have presented the transformation language and the tool Inject/J. The basis of our approach is a meta-model for source-to-source transformation of large object-oriented software. This meta-model provides an abstraction layer that shields the user from the complexity of code-level transformation, i.e. transforming the syntax tree directly. The model provides conceptual analysis and transformations together with their language-specific pre- and post-conditions. Transformation operations are formally defined in terms of their language-dependent primary and secondary code-level transformations. Using conceptual transformations, one can specify transformations that affect large parts of the system, with only local knowledge about the system (e.g. knowledge about certain classes). With the help of detection patterns, one can describe, match, and transform possibly distributed structures in a coherent way. Examples of transformations have been given to illustrate how this all fits together. Our future work includes further improvement of the expressiveness of the transformation language. One improvement is a `wrap` operation that consistently adds code before and after a certain location (e.g. conditional method invocation). Other fields of current work are the evaluation of our tool in industrial case studies and the support for C#.

References

1. The RECODER/Java Homepage. http://recoder.sf.net, 2002.
2. ASF+SDF MetaEnvironment. http://www.cwi.nl/projects/MetaEnv, 2003.
3. AspectJ WWW Page. http://www.eclipse.org/aspectj/, 2003.
4. K. Beck. *Extreme Programming Explained.* Addison Wesley, 1999.
5. J M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proceedings of the ACM Symposium on Software Reusability*, 1995.
6. F. Castor and P. Borba. A Language for Specifying Java Transformations. In *Proceedings of the V Brazilian Symposium on Programming Languages*, 2001.
7. S.R. Chidamber and C.F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 1994.
8. Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach.* PhD thesis, University of Dublin,Trinity College, 2000.
9. Semantic Designs. DMS Software Reengineering Toolkit. http://www.semdesigns.com/products/DMS/DMSToolkit.html, 2003.
10. M. Fowler. *Refactoring – Improving The Design Of Existing Code.* Addison Wesley, 1999.
11. J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *OOPSLA*, 2002.
12. Inject/J WWW Page. http://injectj.sf.net/, 2003.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag, 1997.
14. K.J. Lieberherr. *Adaptive Object-Oriented Software – The Demeter Method.* PWS Publishing Company, 1995.
15. A. Ludwig and D. Heuzeroth. Meta-Programming in the Large. In *Proceedings of Conference on Generative Component-based Software Engineering (GCSE)*, 2002.
16. William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
17. Arthur J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.
18. Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.
19. B. Schulz, T. Genssler, B. Mohr, and W. Zimmer. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS conference*, 1998.
20. O. Seng, T. Genssler, and B. Schulz. Adaptive Extensions of Object-Oriented Systems. In *Proceedings of the IFIP TC2 Working Conference on Generic Programming.* Kluwer, 2002.
21. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* PhD thesis, University of Berne, Switzerland, 2001.

Metaprogramming Library for the C# Programming Language

Gergely Kis[1], József Orosz[2], Márton Pintér[3], Zoltán László[4], and Thomas Genssler[5]

[1] Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
kisg@inf.bme.hu

[2] Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
orosz@mailbox.hu

[3] Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
pmarton@inf.bme.hu

[4] Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
laszlo@iit.bme.hu

[5] Research Center for Information Technologies, Karlsruhe
Research Group Program Structures
genssler@fzi.de

Abstract. As software becomes more and more complex, tool-support for software analysis and transformation is increasingly important. While such tools exist for languages like Java, Smalltalk and C++, the support for C# is poor. In this paper we present Recoder.C#, a library for static metaprogramming of C# programs. Recoder.C# constructs a fully cross-referenced syntax tree and it supports transformation of this syntax tree. The Recoder parser is fully inversive, which means that the original code layout (comments, indentation) is preserved as far as possible. Recoder.C# can be used to build sophisticated analysis and transformation tools, including software metrics and refactorings.[1]

1 Introduction

Throughout its life-cycle, software is continuously modified. It has to fulfil the changing requirements, design flaws have to be found and eliminated, and it has to be optimised to achieve greater performance as well. These changes all include program transformations that can be made either by hand or using a

[1] Work partially supported by the European Community under the IST programme – Future and Emerging Technologies, contract IST-1999-14191 – EASYCOMP. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

software tool. Although transforming by hand is simple and suits most cases, it is rather error prone and time consuming. Considering the current size of software and the fact that modifications have to be realised fast enough to suit business requirements, we can say, it is almost impossible to execute the necessary transformations without tool support. Such tools are already used for projects in many different programming languages like Java, Smalltalk, C and C++.

However, the tool support for the relatively new C# language is poor in this aspect, as there are no effective programs, which can analyse and transform C# code at a high abstraction level.

Our goal was to design and implement a program transformation library for C# that can serve as a core for typical software analysis and transformation tools, such as metrics, visualisations, refactorings, aspect weavers or general-purpose code generators.

To achieve our aim, we need a solution that is able to process programs at higher, structural level than the source code. We found that metaprogramming is an appropriate and capable technique to meet this requirement. Generally there are two approaches to implement metaprogramming. Dynamic metaprogramming modifies the running instance of a software. Our approach, static metaprogramming, operates on the files that make up the program (source or binary), and the transformation process ends before the software is started. In order to implement metaprogramming, a metamodel for the processed language and the associated runtime has to be defined. For each processed program an instance of this metamodel (called the program model) has to be constructed based on the input data (the source files in this case). Based on these program models, the library has to provide interfaces to support analysis and transformation. It should be possible to reflect model changes in the source code. Beyond these basic capabilities real-life scenarios also need source code management and model validation functionality.

We decided to solve the problem by implementing a C# version of the *Recoder for Java* [1] metaprogramming library. The basic theoretical and practical considerations behind the Recoder Framework were published in Andreas Ludwig's and Dirk Heuzeroth's paper [2]. Since the Recoder and its Framework were written in Java, we decided to continue using the Java Platform for the implementation.[2] As a result, it is possible to convert existing Recoder applications easily.

In this paper we describe the architecture and functionality of the library, and picture the future directions of development.

2 Recoder.C#

Recoder.C# is a generic library for program transformations with project and source code management capabilities. It is available [3] under the LGPL license.

[2] The tool itself of course processes C# programs and implements a metamodel for the C# language and runtime.

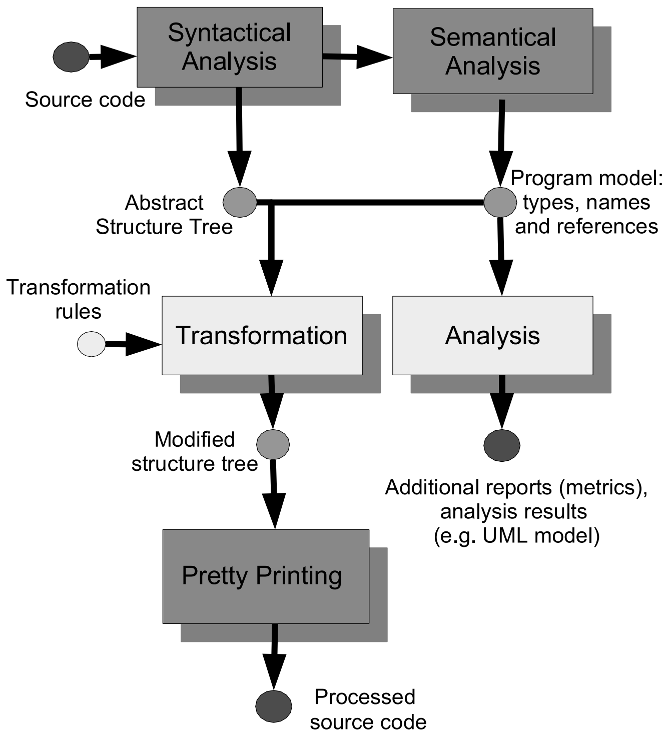


Fig. 1. The architecture of the metaprogramming tool

The architecture of Recoder.C# is shown on Fig. 1. It is based on the existing *Recoder Framework*, thus is almost compatible with the Java version at the interface level. This enables application developers to put their updated products into production in a much shorter release cycle.

The tool's input is the source code of the software to be transformed. The architecture, certainly, makes possible the use of different input formats like machine code or byte code. The task of the syntactical analysis is to create the connection between the abstract syntax and the concrete syntax (the source code). The abstract syntax is usually represented in a tree structure called the Abstract Syntax Tree (AST).

In order to be able to reconstruct the same source file structure, the system needs – unlike normal parsers – to store comments as well. This is not straightforward, as in most languages comments are lexical and not syntactical elements. In the current implementation, Recoder.C# collects the comments in the lexical analysis phase. After the parse is complete the comments are attached to the corresponding AST node using a simple heuristic algorithm.

During the semantical analysis the system computes the additional attributes of the program model using the abstract structure tree (AST) based on the specifics of the actual programming language (in our case C#). By checking these properties computed from the AST one can ensure the validity of the software that is being analysed. Although the implementation is fairly adequate, there exist some limitations. The current version does not handle the operator overload functionality of the language. Work is underway to eliminate this shortcoming.

The Recoder Framework provides transformation operators for the abstract structure tree. The tree changes are executed as transactions. The architecture also makes it possible to define multi-step transformations, where the intermediate steps are not consistent. Currently it is not possible to undo transformations in Recoder.C#, but we are investigating the possibilities.

3 Future Directions

In the previous section we discussed the current state of Recoder.C#. In this section we will enumerate the future directions in the software development cycle.

Preprocessor Support. C# includes a preprocessor. In order to properly support it, we had to rethink the whole metamodel and the architecture of the system. We are currently working on these changes.

Unsafe Extensions. The unsafe extensions of C# provide additional features like pointer arithmetics. Basic support (parsing) is already in place.

Bytecode Analyis. In order to efficiently transform existing programs, there is also need for analyses of code that is only present in the Intermediate Language bytecode. This issue is also being worked on.

4 Conclusion

In the paper we discussed the Recoder.C# metaprogramming library that can be used as a core for complicated software analysis and transformation systems. More Recoder variations exist – each for a different language. These variants try to retain interface compatibility as close as possible, but also incorporate the language-specific features. We overviewed the key features of Recoder.C# and pictured the future directions of the development process.

References

1. The Recoder Homepage: `http://recoder.sourceforge.net`
2. Ludwig, A., Heuzeroth, D.: Metaprogramming in the large. In: 2nd International Conference on Generative and Component-based Software Engineering (GCSE). Also available as LNCS 2177 (c) by Springer. Number 2177, Springer (2000)
3. The Recoder.C# Homepage: `http://recoder-cs.sourceforge.net`

Author Index