Lecture Notes in Computer Science 1870

Pierre Deransart Manuel V. Hermenegildo
Jan Małuszyński (Eds.)

Analysis and Visualization Tools for Constraint Programming

Constraint Debugging

Springer

Lecture Notes in Computer Science 1870
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Pierre Deransart Manuel V. Hermenegildo
Jan Małuszynski (Eds.)

Analysis and Visualization Tools for Constraint Programming

Constraint Debugging

Volume Editors

Pierre Deransart
INRIA-Rocquencourt, Project “Contraintes”
B.P. 105, 78153 Le Chesnay Cedex, France
E-mail: Pierre.Deransart@inria.fr

Manuel V. Hermenegildo
Technical University of Madrid, School of Computer Science
28660 Madrid, Spain
E-mail: herme@fi.upm.es

Jan Małuszynski
University of Linköping
Department of Computer and Science
581 83 Linköping, Sweden
E-mail: janma@ida.liu.se

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Analysis and visualization tools for constraint programming :
constraint debugging / Pierre Deransart ... (ed.). - Berlin ;
Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ;
Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; 1870)
ISBN 3-540-41137-2

CR Subject Classification (1998): D.1, D.2.5, D.3.2-3, I.2.3-4, F.4.1, I.2.8

ISSN 0302-9743
ISBN 3-540-41137-2 Springer-Verlag Berlin Heidelberg New York

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

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10722311 06/3142 5 4 3 2 1 0

Foreword

Coordinating production across a supply chain, designing a new VLSI chip, allocating classrooms or scheduling maintenance crews at an airport are just a few examples of complex (combinatorial) problems that can be modeled as a set of decision variables whose values are subject to a set of constraints. The decision variables may be the time when production of a particular lot will start or the plane that a maintenance crew will be working on at a given time. Constraints may range from the number of students you can fit in a given classroom to the time it takes to transfer a lot from one plant to another. Despite advances in computing power, many forms of these and other combinatorial problems have continued to defy conventional programming approaches.

Constraint Logic Programming (CLP) first emerged in the mid-eighties as a programming technique with the potential of significantly reducing the time it takes to develop practical solutions to many of these problems, by combining the expressiveness of languages such as Prolog with the computational power of constrained search. While the roots of CLP can be traced to Monash University in Australia, it is without any doubt in Europe that this new software technology has gained the most prominence, benefiting, among other things, from sustained funding from both industry and public R&D programs over the past dozen years. These investments have already paid off, resulting in a number of popular commercial solutions as well as the creation of several successful European startups.

This book is about DiSCiPl, a two-and-a-half-year European project aimed at paving the way to broader adoption of CLP. DiSCiPl stems from the observation that, while CLP can significantly reduce the time it takes to develop practical solutions to complex combinatorial problems, doing so often involves a lot of tinkering and deep insight into the innerworkings of the language and its underlying search mechanisms. The objective of the project, which was launched in late 1996 in the context of the European Research Program in Information Technology (ESPRIT), was to research and validate new concepts and tools aimed at significantly facilitating the development and refinement of CLP programs, with a special focus on "Constraint Debugging". "Debugging" here is to be interpreted in the broad sense and includes both

"correctness debugging", namely ensuring that a CLP program properly captures all aspects of a given problem, and "performance debugging", which has to do with analysing and fine-tuning performance of CLP programs.

DiSCiPl brought together some of the best brains in the field in Europe, combining participation of four leading research organisations (INRIA in association with ERCIM, UPM, the University of Linköping and the University of Bristol), two CLP vendors (Cosytec and PrologIA) and two solution providers (ICON and OM Partners). The results, which are presented in this book, include a novel methodology for CLP debugging together with a rich collection of new debugging techniques. At the time of writing, some of these techniques have already found their way into a number of popular CLP packages, making their benefits available to a sizable user population. A nice feature of the book is its discussion of user cases, detailing these benefits. Beyond its more immediate impact, the DiSCiPl project has also produced significant theoretical results that open the door to new and exciting avenues for future research.

It has been a privilege and a pleasure to work with such an enthusiastic group of people from the very inception of their project and to see many of their results so quickly made available to the user community. I hope that you will enjoy reading this book as much as I have.

January 2000

Norman M. Sadeh
European Commission

Preface

This book is the first one entirely dedicated to the problem of **Constraint Debugging**. It presents new approaches to debugging for the computational paradigm of **Constraint Programming** (CP).

Conventional programming techniques are not well suited for solving combinatorial problems in industrial applications like scheduling, decision making, resource allocation, or planning. Constraint Programming offers an original approach allowing for efficient and flexible solving of complex problems, through combined implementation of various constraint solvers, expert heuristics, and programmed search. As an emerging software technology, Constraint Programming is relatively young compared to other languages, technologies, and paradigms. However, it has already produced convincing results and its applications are increasingly fielded in various industries.

One of the remaining shortcomings of CP technology is that it is still somewhat difficult to use. This is due to the intrinsic complexity of the problem areas tackled and to the comparatively sophisticated solutions offered by this technology. These difficulties can be overcome by a combination of adequate training and the availability of effective debugging techniques and environments adapted to the particular characteristics of the paradigm. In fact, one of the main features of CP is a new approach to software production: the same program is progressively improved at each step of the development cycle, from the first prototype until the final product. This makes debugging one of the cornerstones of CP technology.

Debugging is considered here in a broad sense: it concerns both validation aspects (to build a correct application) as well as methodological aspects (to find the best solution to a problem by gaining a better understanding of constraint solver behavior). To satisfy these objectives, tools must locate and explain bugs, and graphical tools must help in the process of interpreting program behaviour and results. The debugging tools of the commercial CP systems are often ineffective in industrial situations, and the debugging techniques originating from imperative languages are mostly inapplicable to CP. The main reason is that the huge numbers of variables and constraints make the computation state difficult to understand, and that the non-deterministic execution drastically increases the number of computation states which must be analysed.

This book contains most of the results of the DiSCiPl project. DiSCiPl (Debugging Systems for Constraint Programming) is a recently completed European (IT4) reactive Long Term Research project which had been running for over two and a half years, from October 1996 to June 1999, with four academic (INRIA-Rocquencourt, manager, with ERCIM, UPM, University of Linköping, University of Bristol) and four industrial partners (Cosytec, PrologIA, ICON, OM Partners). The objectives were to develop the theory of constraint debugging and to create tools to help the programmer during all phases of development.

DiSCiPl has produced a good number of results at both the theory and implementation levels. The new theoretical results in debugging have been cast into the form of a practical "DiSCiPl debugging methodology". These practical developments have produced enhanced versions of industrial and academic Constraint Logic Programming (CLP) platforms (specifically Prolog IV, Chip++5.2.1, GNU-Prolog and Ciao/Prolog) with new, rich debugging capabilities.

This book is a first attempt to give a unified view of "constraint debugging" and it presents the results of the DiSCiPl project in a more comprehensive manner. Technical details can be found in various publications originated from the project or in the public deliverables available at the URL http://discipl.inria.fr.

The DiSCiPl project allowed making significant progress towards understanding the problem of constraint debugging and produced a good number of results and tools. It however did not close the topic. On the contrary we believe that it has opened a field, showing that debugging is an essential part of the process of constraint programming. Only some aspects of debugging have been explored and only still incomplete tools have been produced in the limited duration of the project. They are starting points. Specific suggestions for future work in the area of constraint debugging are presented in different chapters of this volume.

The book consists of an introduction and three parts, each of them composed of several chapters. Most of the chapters can be read independently. The introduction (chapter 1) presents the "DiSCiPl debugging methodology" and explains how all chapters are related.

- Part 1: **Correctness Debugging** Five chapters presenting techniques and tools for finding the reasons for incorrect behaviour of a constraint program. Most of them use assertions and static analysis techniques.
- Part 2: **Performance Debugging** Seven chapters presenting visualization tools which facilitate understanding of the search space and of the constraint propagation. They facilitate finding the reasons for inefficient execution (performance debugging) and they may also contribute to correctness debugging.

– Part 3: **User cases** One chapter which presents feedback from the use of some of the debugging tools in an industrial context.

The research presented in this book has been influenced by the feedback obtained during the project reviews from the the reviewers Dominique Boliganano and Seif Haridi, and the project officer, Norman M. Sadeh. We are very grateful to all of them. The members of the Advisory Committee followed the project giving us very useful and sometimes very enthusiastic feedback.

In addition to European ESPRIT LTR project DiSCiPl # 22532 this work has been partially supported by Spanish CICYT Projects TIC97-1640-CE and TIC99-1151.

April 2000

P. Deransart
M. Hermenegildo
J. Małuszyński

Table of Contents

List of Contributors

A. Aggoun
COSYTEC SA
4, rue Jean Rostand
91893 Orsay Cedex
France

C. Aillaud
Gemplus
Avenue Pic de Bretagne
Parque d'activité de Gemenos
BP100
13881 Gemenos Cedex
France
Christophe.Aillaud@gemplus.com

N. Beldiceanu
COSYTEC SA
4, rue Jean Rostand
91893 Orsay Cedex
France

F. Benhamou
IRIN, Université de Nantes
BP 92208
44322 Nantes Cedex 3
France
benhamou@irin.univ-nantes.fr

E. Bourreau
BOUYGUES- Direction des Technologies Nouvelles
1, av. Eugène Freyssinet
78061 St.-Quentin-en-Yvelines Cedex
France
ebourrea@challenger.bouygues.fr

P. Bouvier
Société PrologIA
Parc Technologique de Luminy
Case 919
13288 Marseille Cedex 9
France
bouvier@prologianet.univ-mrs.fr

F. Bueno
School of Computer Science
Technical University of Madrid
28660 Madrid
Spain
bueno@fi.upm.es

M. Carro
School of Computer Science
Technical University of Madrid
28660 Madrid
Spain
mcarro@fi.upm.es

T. Cornelissens
OM Partners
Michielssendreef 42
2930 Brasschaat, Belgium
tcornelissens@ompartners.com

P. Deransart
INRIA-Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Pierre.Deransart@inria.fr

W. Drabent
Institute of Computer Science,
Polish Academy of Sciences
ul. Ordona 21
01-237 Warszawa
Poland
wlodr@ida.liu.se

V. Dumortier
OM Partners
Michielssendreef 42
2930 Brasschaat, Belgium
vdumortier@ompartners.com

G. Fabris
ICON s.r.l.
Viale dell'Industria 21
37121 Verona, Italy
gfabris@icon.it

G. Ferrand
LIFO
BP 6759
45067 Orléans Cedex 2
France
Gerard.Ferrand@lifo.univ-orleans.fr

F. Goualard
IRIN, Université de Nantes
BP 92208
44322 Nantes Cedex 3
France
goualard@irin.univ-nantes.fr

M. Hermenegildo
School of Computer Science
Technical University of Madrid
28660 Madrid
Spain
herme@fi.upm.es

C. Laï
Société PrologIA
Parc Technologique de Luminy
Case 919
13288 Marseille Cedex 9
France
lai@prologianet.univ-mrs.fr

J. Małuszyński
Linköpings universitet
Department of Computer and Information Science
581 83 Linköping
Sweden
janma@ida.liu.se

F. Nanni
ICON s.r.l.
Viale dell'Industria 21
37121 Verona, Italy
nanni@icon.it

P. Pietrzak
Linköpings universitet
Department of Computer and Information Science
581 83 Linköping
Sweden
pawpi@ida.liu.se

G. Puebla
School of Computer Science
Technical University of Madrid
28660 Madrid
Spain
german@fi.upm.es

H. Simonis
COSYTEC SA
4, rue Jean Rostand
F 91893 Orsay Cedex
France
simonis@cosytec.fr

A. Tessier
LIFO
BP 6759
F 45067 Orléans Cedex 2
France
Alexandre.Tessier@lifo.univ-orleans.fr

A. Tirabosco
ICON s.r.l.
Viale dell'Industria 21
37121 Verona, Italy
tirabosco@icon.it

Debugging of Constraint Programs: The DiSCiPl Methodology and Tools

This introduction gives a general perspective of the debugging methodology and the tools developed in the ESPRIT IV project DiSCiPl *Debugging Systems for Constraint Programming*. It has been prepared by the editors of this volume by substantial rewriting of the DiSCiPl deliverable *CP Debugging Tools* [1].

This introduction is organised as follows. Section 1 outlines the DiSCiPl view of debugging, its associated debugging methodology, and motivates the kinds of tools proposed: the assertion based tools, the declarative diagnoser and the visualisation tools. Sections 2 through 4 provide a short presentation of the tools of each kind. Finally, Section 5 presents a summary of the tools developed in the project. This introduction gives only a general view of the DiSCiPl debugging methodology and tools. For details and for specific bibliographic references the reader is referred to the subsequent chapters.

1 The DiSCiPl View of Debugging

The work performed in the project has addressed two main categories of CP debugging:

- *correctness* debugging, and
- *performance* debugging.

Correctness debugging aims at locating program constructs that cause that the computed answers are different from those expected. In particular this concerns syntax errors, wrong answers, missing answers, mode violations, non termination, and several kinds of unexpected behaviour. Detection of inconsistencies in input data is another topic of correctness debugging.

Performance debugging aims at identification of the reasons for poor efficiency of computations, such as inadequate labelling strategy or poor constraint propagation. The overall goal of performance debugging is to shorten the time for finding feasible and/or optimal solutions.

The subsequent sections summarise the approaches pursued in DiSCiPl for addressing correctness debugging and performance debugging.

1.1 Correctness Debugging in DiSCiPl

DiSCiPl pursued two different approaches to correctness debugging:

- static debugging,
- debugging based on run-time symptoms

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 1–20, 2000.

Static Debugging. *Static debugging* aims at finding errors without executing the program. In DiSCiPl static debugging is based on static analysis of the program and on automatic checking of assertions.

Automatic checking of assertions may be able to prove correctness of the program with respect to formally specified requirements concerning certain properties of the program that should hold in *all* computations. For example, a formal requirement may specify call and success patterns of a predicate in all computations of the program. If a program is not correct w.r.t. a given specification, any attempt to prove it will fail.

Static analysis techniques make it possible to infer automatically certain properties of the program. The inferred properties may or may not conform to user expectations, or to a priori given specification describing the expectations. In the latter case the discrepancy between the inferred and the expected/specified properties is called an *abstract symptom*. Thus, abstract symptoms may be found manually, by inspection of the results of static analysis, or automatically, by comparison of the assertions describing expected properties with those generated by static analysis. Existence of an abstract symptom shows that the program is not correct w.r.t. the specification of the expected properties.

The process of locating a construct causing the abstract symptom will be called *static diagnosis*. This construct will also cause a failure in any attempt to prove correctness of the program with respect to the specification of expected properties, and can be located in such an attempt.

A crucial issue in static debugging is how to design a language for expressing specifications and results of static analysis. The language should on one hand allow expressing complex properties such as those inferred by static analysis and on the other hand should be easy to understand by the user. The statements of the language will be called *assertions*. Thus, assertions should provide a basis for two-way communication between the user and the static debugging tools.

It should be noticed that the use of assertions is not restricted to static analysis and correctness proofs. In some DiSCiPl tools they are also used as tests checked during the execution of the program or for selection of particular data for inspection by the user during execution visualisation.

Debugging Based on Run-time Symptoms. This approach concerns the case when in some execution the program behaves in some way which is different from that expected by the user. Such a discrepancy between the expected and the actual behaviour of a program in a single computation is called a *run-time symptom*. Debugging based on run-time symptoms aims at finding the cause of a run-time symptom that occurred in a single execution. In the case of correctness debugging the symptom can usually be linked to an atom, a predicate or a clause. This may be achieved by analysis of the computation where the run-time symptom appears. A run-time symptom, such as a wrong answer or a missing answer, can also be characterised in terms

of declarative semantics. Thus, at least for some run-time symptoms, debugging can be based either on the declarative semantics or on the operational semantics. Both possibilities have been pursued in DiSCiPl.

An approach to debugging based on declarative semantics known as *declarative diagnosis* [12] makes it possible to locate errors responsible for wrong and missing answers. The method has been further developed in DiSCiPl.

On the other hand, some of the DiSCiPl visualisation tools (designed primarily for performance debugging) facilitate analysis of single computations, hence debugging based on operational semantics.

1.2 Performance Debugging

Poor performance is often caused by modelling of the problem in an inadequate way or by adopting an inadequate search strategy. Often, such an error cannot be identified just at the level of a single predicate or constraint. Unfortunately, there are no general methods for finding reasons for poor performance. Therefore, it is necessary to provide tools which help the user to understand (very) complex computations. Such tools should be able to present a complex computation at various levels of abstraction in a form easy to understand by the user. Three aspects of particular importance are:

– presentation of the control of the execution,
– presentation of the constraint store at different points in the execution
– presentation of global constraints.

Tools must also allow to re-do or modify computations at some point with the purpose of exploring different strategies.

The DiSCiPl approach to performance debugging was based on the guidelines listed above. Several visualisation techniques proposed and implemented in the project will be surveyed below and will be discussed in more detail in separate chapters. It should be noted that these are general tools which help the user to understand the execution of the program, and thus, in addition to performance debugging, they may also be of interest for correctness debugging.

1.3 The DiSCiPl Debugging Methodology and Tools

We now present a general view of the proposed DiSCiPl debugging methodology and show how it is supported by the tools developed in the project. The proposed debugging methodology consists of (possibly interleaved) successive steps of correctness debugging and performance debugging. Location of an error results in modification of the program. The new version may still be subject to debugging.

The proposed methodology is illustrated in Figure 1. At each stage of the program development process we have an actual version of the program,

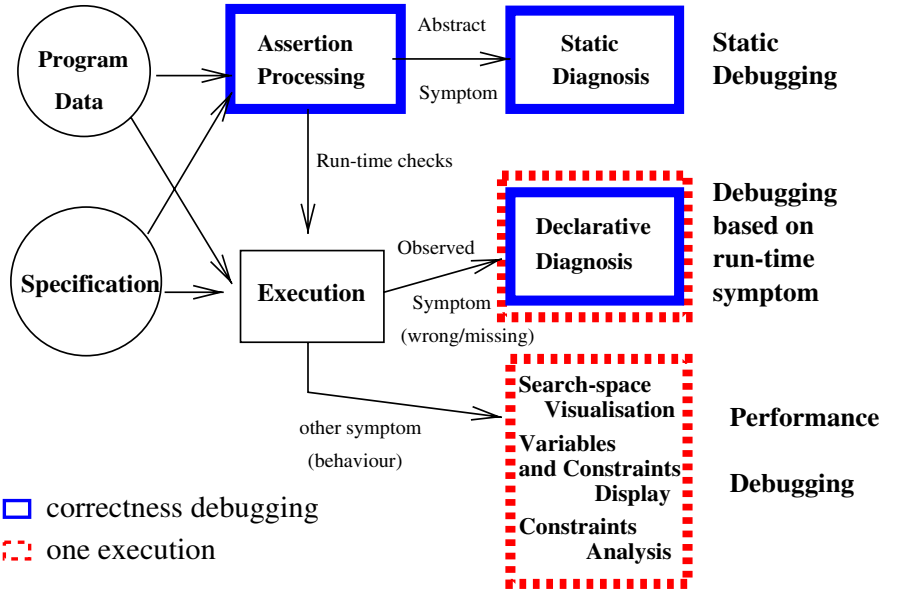


Fig. 1. DiSCiPl Debugging Methodology

(possibly) some input data and a (possibly empty) specification expressed in the assertion language.

The underlying debugging methodology is as follows. The first objective is, understandably, program correctness. Correctness is dealt with at a first stage at compile-time. We distinguish here the *assertion processing* phase and the *static diagnosis* phase.

Two different approaches to the assertion processing phase have been used in DiSCiPl. The first one is an extension of the classical verification techniques to the case of constraint logic programs. In this approach, a complete specification of the program is required and verification aims at actually proving that the program satisfies such specification. Failure in proving the specification may locate program constructs which are responsible for bugs. The second approach is based on first inferring properties of the program by means of static analysis and then comparing such properties with the existing specification. If the inferred properties are incompatible with some part of the specification then an abstract symptom is found. The advantage of this second alternative is that a complete specification of the program is not required, though the more complete such specification is, the more bugs can be automatically detected.

In case abstract symptoms are discovered in the assertion processing phase, they should be corrected before executing the program. After correcting the program, assertion processing must be performed again in order

to check whether abstract symptoms still remain in the corrected version of the program. Though the assertion processing tools often identify the cause of an abstract symptom, it is also convenient to have automatic tools for static diagnosis, as indicated in the figure. The intention is to detect bugs as early as possible, i.e., during compilation or even editing. This can only be achieved by (semi-) automatic analysis of the (not necessarily completely developed) program in the presence of some (approximate) specifications. An example of such techniques is type checking, which has long been proved to be useful for this purpose. Our approach introduces a framework for working with properties that will be more general than classical type systems.

The fact that no more abstract symptoms are detected does not imply that no bugs are left in the program. At this point, program testing in the classical sense has to be performed. However, the methodology also aids the programmer in this respect. During the assertion processing phase, programs may optionally be automatically annotated with run-time checks for those requirements which turned out to be impossible to prove in the assertion processing phase. This will allow detecting further violations of the specifications during program execution on sample input data. The result is detection of run-time symptoms. These have to be diagnosed and the detected errors have to be corrected. Since run-time tests introduce substantial overhead in program execution, the annotation with run-time checks is performed only upon user's request and possibly temporarily, i.e., only until the behaviour of the program can be assumed to be correct.

Once correctness is sufficiently established, it is time for performance debugging. The visualisation tools developed in the project make it possible to present abstract views of a single computation. In particular, they will be used to examine those computations that are inefficient in finding expected solutions.

Figure 1 gives a general idea about the tools needed to support the proposed methodology. They include tools like assertion processors, static diagnosers, declarative diagnosers, and various kinds of visualisers. Several tools of this kind have been implemented in the project on various platforms. They are surveyed in the forthcoming sections and discussed in more detail in the remaining chapters of the book.

2 The Assertion-Based Tools

The static debugging tools developed in DiSCiPl are based on assertions. We first briefly discuss the language of assertions and then the static debugging tools based on it.

2.1 The Language of Assertions

Assertions are used for describing selected aspects of the intended or/and the actual behaviour of the program. Thus, they specify some requirements about all answers or computations, or they describe some properties that are satisfied by all answers (or by all computations) of the program.

There is a trade-off in the design of an assertion language: the more expressive the language of assertions is, the more subtle properties of programs can be described. However, this also causes increased difficulties in verification and/or inference of such subtle properties. Also, a complex assertion language could be difficult to understand by the user.

The choice of a suitable assertion language was an important design decision that provided a common basis for development of various tools on different platforms. An extensible assertion language has been designed and is in use in several tools developed in the project. The language is described in more detail in Chapter 1. Here, we outline only some principles of its design and we give a few examples of assertions.

There are two main kinds of assertions, those which relate to predicates, called *predicate assertions* and those which relate to program points (*program-point assertions*). Predicate assertions can be classified into three categories: (1) assertions describing success states of a predicate, sometimes called *postconditions*, (2) assertions describing call states of a predicate, sometimes called *preconditions*, (3) assertions describing properties of the whole computation of a predicate (a sequence of states).

Predicate assertions link with a predicate some properties which should hold, respectively, in all call states, all success states or in all computations. For example, the assertions:

```
:- calls p(X,Y) : ground(X).
:- success p(X,Y) => list(Y).
:- comp p(X,Y): (ground(X), var(Y)) + not_fail.
```

state, respectively, that in all calls to `p`, the first argument must be ground (i.e., a term without variables), in all success states for `p` the second argument must be a list, and that every call to `p` with the first argument ground and the second a variable should not fail.

The language of assertions is extensible in the sense that its definition does not restrict a priori the properties to be referred to by the assertions. Properties can be predefined predicates of the language (like `ground` and `var` above) but can also be defined by the user. Chapter 1 discusses general aspects of using the DiSCiPl assertion language for program validation and debugging.

2.2 The Tools

We now discuss briefly the DiSCiPl tools supporting static debugging. They include: the Generic Preprocessor for CLP Debugging and its two instances

CiaoPP (for the Ciao Prolog system) and CHIPRE (for CHIP), described in Chapter 2, the Prolog IV assertion tool (Chapter 3), and the static type-based diagnoser of CHIP discussed in Chapter 4 which has been also ported to Calypso.

We assume that a program is given together with a (possibly empty) set of assertions describing some required properties of the program (to be called *check assertions*).

Ideally, we would like the assertions to be automatically *checked*. That is, we would like to have a tool that receives a program and a set of check assertions as input and returns, for each of the assertions, one of the following three values:

- **proved:** the requirement holds in every computation. This is the most favourable situation. It indicates that the requirement is verified.
- **disproved:** the requirement is proved not to hold in some computations. This means that something is wrong in the program and we should locate the program construct responsible for that.
- **unknown:** the tool is unable to prove or disprove the requirement. If this may happen the tool is called *incomplete*.

Such automatic checking of assertions is one of the basis for static debugging in DiSCiPl. Though the tools used are incomplete, in many cases they are able to prove or to disprove the check assertions provided by the user. For a disproved set of check assertions, the constructs responsible for incorrectness of the program are identified in the static diagnosis process. Even in "don't know" cases it is often possible to locate fragments of the programs where violation of the check assertions is not excluded. This is often a useful warning.

It is often the case that no check assertions are provided, or the verification of the provided ones gives a "don't know" answer. The DiSCiPl static debugging tools address this problem in different ways.

The principle of the preprocessors (CiaoPP, CHIPRE) is fully automatic operation that for a given program and (possibly empty) set of check assertions finds as many abstract symptoms and locates as many errors as possible. The preprocessor uses *abstract interpretation* [5] to infer actual properties of a given program which are then written in terms of assertions. Automatic inspection of such inferred assertions allows detecting irregularities, like empty types of predicates, which are often good indicators of bugs in the program. The tool also compares the inferred assertions with the existing check assertions and with the assertions describing relevant properties of the library predicates and reports the discovered abstract symptoms. This sometimes may be sufficient to locate errors and generate warnings. For an assertion which cannot be proved or disproved run-time tests may be included in the program. It is important to mention that the preprocessor may detect a good number of bugs without any user-provided check assertions.

In the Prolog IV assertion tool the focus is on verification of check assertions. In order to use it, one has to augment the program with check assertions describing expected call states and success states of all program predicates. The assertions are to be constructed from a small set of basic properties including atomic constraints of Prolog IV and a few other properties. The tool attempts to verify the assertions and reports on the errors located during the verification and also on missing and inconsistent assertions. Run-time tests are inserted for assertions which were neither proved nor disproved during the verification. In contrast to the preprocessors this tool is not able to handle programs without check assertions.

The type-based diagnoser is an interactive tool for locating causes of abstract symptoms. The diagnosis is done by searching for verification failures, thus using a principle similar to the above discussed tool. An important difference is that the check assertions needed are not assumed to be given a priori but are interactively requested "by need". The process is started and controlled by a discovered abstract symptom. The session is preceded by static analysis of the program. The analyser infers types of all predicates. They may be inspected by the user and the session starts when one of them is identified as an abstract symptom. (Alternatively a symptom can be obtained by comparison of the inferred types with check assertions, if the latter are given). The requests of the diagnoser concern expected types of program predicates. They may be answered by referring to the types inferred or by providing different ones.

3 The Declarative Diagnoser

Declarative diagnosis is a technique for locating errors in a program on the basis of run-time symptoms that appear in a single terminating computation. The diagnosis session may be started if the outcome of a terminated computation is a symptom. The key idea is that the analysis of the computation is partly automated. The system queries the *oracle*, which is normally the user, about intermediate results of the analysed computation. The oracle has to decide whether the results shown at each step of the diagnosis are symptoms or not. In addition, it has to tell the system what would be the expected result if the one shown is a symptom. Oracle answers are used by the diagnosis algorithm to control the search for the error.

It should now be clear why the diagnosis is called declarative: the queries concern only the results, that is, they refer to the declarative semantics of the program, while the operational aspects of the analysed computation are hidden from the oracle. This allows to locate errors in the program without having to deal with control aspects, which are hidden in the diagnosis algorithm.

The symptoms dealt with by a declarative diagnosis divide naturally in two categories:

– an *incorrectness symptom* is a wrong answer. More precisely, the computed answer for a given goal g is a constraint c. It is an incorrectness symptom if c has some solutions not expected by the user.
– an *incompleteness symptom* is observed if some specific solution expected by the user is not given by any of the computed answers.

Declarative diagnosis of each kind of the symptom is done by different algorithms.

The declarative diagnoser developed in DiSCiPl is described in Chapter 5. The main difficulty with the declarative diagnosis technique is that the intermediate results may be difficult to understand by the human oracle. The visualisation and abstraction techniques developed in the project may be helpful in this respect.

4 Visualisation Tools

Sometimes the tools discussed so far cannot detect the source of an incorrect or missing answer in a program, or the program shows an unsatisfactory performance whose source cannot be detected by static analysis-based tools. The latter behaviour is usually intimately related with the operational semantics of the language, and cannot be uncovered by tools based on declarative semantics. In the former case, sometimes an extensive exploration of the runtime behaviour of the program (possibly with the aid of some intuitive, high-level representation of such behaviour) is of great help, allowing the programmer to grasp the problems present in the execution. In this section we will survey tools facilitating such an exploration. They are intended to be used when the performance of the program is unsatisfactory and also when the program shows incorrectness symptoms which are difficult to handle with static or declarative debugging tools.

Most visualisation and runtime debugging tools are based on representations of the execution profile of constraint programs using a Prolog-like selection rule, and thus depict an LD tree. However, classical debuggers (i.e., those customarily used for imperative languages, and even for Prolog) fall altogether short when it comes to CLP, and new approaches are needed: the implicit selection rule, the use of backtracking, the constraint propagation, and the (encapsulated) labelling strategies of CLP have to be taken into account and somehow reflected in the representation of the execution, for in many cases the culprit of an unexpected behaviour can be traced back to a wrong selection affecting one or more of these features.

Additionally, classical debuggers for imperative languages show values of variables in a straightforward way, since they are perceived as boxes holding their values, with no explicit relationship whatsoever with each other. However, values of variables in CLP are not simple items, but logical variables,

and they can hold not only definite values, but ranges of values (either finite or infinite, depending on the domain of the language), and the basic elements in the domain can be integers, reals, boolean values, strings, etc., again depending on the domain. The actual range of a variable is updated as execution proceeds, being usually narrowed as the system advances towards a final solution, or being widened, when the system backtracks to search using an alternative path. These changes in the ranges of the variables are of utmost importance to understand an execution and the sources of a possible low performance.

Variables in CLP are also related among them by setting up constraints. From the viewpoint of the user, this is performed just by stating the equations in the source language. But internally, the representation and management of the constraints is very involved, and equivalent constraints differently expressed (for example, over-constraining some variables) can greatly affect the execution time. The possibility of visualising the constraint store and its evolution allows the programmer to perceive the relationship among the source code and the behaviour of the constraints.

Additional facilities which would be of great help are the possibility of projecting the constraint store at a given point on some variables (for example, on the variables of a clause). Moreover, current CLP systems have evolved to tackle large, real world problems, which has forced the introduction of complex, specialised constraints, such as the cumulative constraints [2], for which *ad hoc* representations are better suited. Visualisation of an instance of the cumulative constraint should show its evolving profile at runtime.

In any of the above cases, large executions pose a general problem: the amount of information to be treated or displayed cannot be dealt with using standard techniques, and special means of *abstraction* have to be applied. The aim is to allow the user to focus on interesting properties of the program, and to abstract automatically from irrelevant details.

In the next sections we will show how the above mentioned needs are addressed by the tools developed in the project. Some of the tools are specialised for a particular CLP system, while others exemplify more general techniques amenable to be adapted to a variety of systems.

4.1 Showing Control Features

Although one of the basic and very interesting properties of constraint programs is that it can generally be understood declaratively, i.e., without looking at control aspects, a concrete control strategy which drives the execution flow (and, thus, the search) implicitly exists in the evaluation engine. Sometimes a lack of understanding of how this control operates makes a program difficult to debug from the performance point of view. This control has two important components: the search determined by the program and the search in labelling. The control in the propagation of constraints, although permitted by some systems, does not fall in the same category as the flow control. While

it is true that different propagation schemes affect the runtime constraints and, therefore, may ultimately change the *shape* and performance of the execution, the relationship between the settings of the constraint propagation control and the shape of the search tree (or, more precisely, the labelling tree) are too distant to be considered intimately related and explored in the same way.

The Box Model and its Appropriateness for CLP. Classical Prolog debuggers are generally based on the well-known "box model" [3]. This is an execution model where a procedure invocation is seen as a box, with unidirectional input- and output gates, named *ports*. These ports are associated with events happening to this call/box: entering the box for first time (i.e., calling the predicate), exiting with success, reentering to try another choice (i.e., redoing), and exiting with failure. Other ports can be defined—and turn out to be necessary to represent the operational behaviour of CLP systems. For example, suspended constraints waiting for instantiation, if they are to be seen as regular goals, need a sort of *suspend* and *awake* port. This complicates the representation of the execution, since the control and constraint solving parts of the system are mixed in a single execution flow. This approach possibly complicates a text-based debugger too much, even if the classical commands to break the execution at some point, and to skip or dive into parts of the program are available. Further enhancements, such as commands to display the state of the stacks or to locate the current execution point in the overall picture are an important aid, but still the problem of big executions and representation of variables (which is a relatively simple projection in Prolog) and constraints remain to be solved.

It is possible to make use of graphics in order to represent the boxes more intuitively; a first approach would be to literally display them, ones inside the others, and provide a zoom in/out facility to navigate in them. However, the depiction becomes very complicated when there are more levels of nesting, and probably this representation is only of pedagogical interest.

Showing Control as a Tree. Another well-known idea is to visualise the resolution tree. This representation lends itself to a variety of further refinements and additions which makes it quite appealing, despite its initial simplicity.

One of the strong points of a graphical depiction of control is that the "big picture" of the execution is unveiled, so that the programmer can look at the execution globally, both in the programmed search and in the labelling part of finite domain solvers. The programmed search, despite being the core search technique in Prolog programs, is, in general, second to the labelling procedures in CLP. This labelling can, in fact, be seen as a specialised, data-driven (often including non-trivial heuristics) search. Moreover, it is not fixed, since several constraint logic programming systems (as CHIP or Prolog IV), allow the programmer to provide parameters to the labelling predicate which tailor its behaviour to the application at hand. If a visualisation of the

enumeration is available, the user can control at each node how the chosen strategy behaves. This, combined with visualisation of the range of variables, will provide the programmer with a good view of how the search space (as determined by the domain of the variables at each execution point) is being reduced.

Thus, it appears useful to develop tools which show globally how the execution proceeded, both from the point of view of the program search and of the labelling steps. Since it is interesting to show which variables are involved in a given node (predicate call), it appears advantageous to interface tools aimed at this representation with the control view. We will talk about these tools in the next sections.

In view of the above, a feasible general strategy to implement control-related depictions, and to interface them with other visualisation tools, is the following:

A generic search tree visualiser should be able to show graphically the nodes of both the programmed search and the labelling search trees.

- In the programmed search view, a node should be shown per predicate call, including predicate names and, if possible, depiction of the source code and of the runtime data (i.e., the values of the variables). For some domains, as the Herbrand domain, this can be taken care of directly by the tree visualiser, but as the complexity of the domains increases, it is probably a better design choice to interface the raw tree visualiser with other tools aimed at the visualisation of variables. Several control actions should be possible while displaying this tree, namely: stopping the execution, stepping forward and backward, and abstracting parts of the execution in order to avoid overwhelming the programmer with too many details (see below). In addition, it should be possible to obtain selectively (e.g. by clicking on the nodes of the tree) a (possibly graphical) view of variables and of the constraint store at the execution points corresponding to the nodes of the tree.
- For the labelling search view, the nodes of the tree correspond to the choices performed within the labelling procedure. This, combined with a variable domain visualisation, which highlights how the current domain of the variables is being narrowed, provide the programmer with a good view of how the search space is reduced. Abstractions techniques can be applied, in a similar way to the programmed search view.

4.2 Showing Values of Variables

The values of variables drive the execution, and obtaining them is its ultimate objective. Knowing them at runtime is a comparatively easy task in the case of traditional languages, but in the realm of constraint languages the situation becomes more involved: definite values are now transformed into constraints

which relate the values of the variables and which restrict the possible range of values a variable can take.

In principle all the variables in the program could be visualised (i.e., the store itself could be shown as a collection of variables), but it is clear that in most cases this would overwhelm the programmer with an unwanted amount of information. A possibility which reduces the number of variables to take into account is to select those variables which are reachable from a given program point; this can be done by hand (inspecting the program) or by using special tools. Alternatively, some variables can be marked especially in the source program with annotations which do not change the meaning of the program, but which are understood by programs aimed at debugging.

Depicting the variables themselves needs a way of representing their domains. While for some constraint domains no satisfactory solution has been developed so far (e.g., for linear constraints), for other, such as finite domains, reasonably easy to understand depictions can be used. Textual representation of the domains, or graphical depictions such as that developed by Micha Meier [9] for the Eclipse system [7] will be used in the tools described in the next chapters. In particular, the graphical representation in [9], based on assigning a dot (c.f., square) to each possible value of a variable, which is therefore shown as a collection of dots or squares, is at the same time compact and amenable to be used to represent the relationships among variables and its history in time.

4.3 Showing Constraints

Constraint debugging refers to the process of debugging programs by examining the constraint store, as opposed to examining the structure (and the execution) of the program. A constraint-oriented view is helpful both in correctness debugging, since by inspecting the store the programmer may detect wrong or missing constraints, and in performance debugging, because the structure of the constraints determines the propagation of updates inside the solver, hence the number of internal operations performed. A more in-depth discussion of this issue can be found in several related chapters.

The variable visualisation tools mentioned above do not give any direct insight into the relationships or mutual influence among variables. As the values of variables are updated by adding constraints (which restrict the domain of the variables, or relate different variables), exploring which constraints are active at a given point, and which constraints were used to perform propagation at some point, is also very interesting. In general, it is useful to show some or all the constraints in which a subset of variables are involved (thus projecting the store over these variables). Obvious constraint representations include text-based displays of (projections of) constraints using the source language, but, as it happened with values of variables, this may not always be appropriate if too many constraints or variables are involved. Additionally,

a source-based representation of constraints is usually very difficult to understand intuitively, especially when the number of constraints grows beyond some threshold.

It is possible to develop graphical representations of the relationships among variables: an appealing possibility is to allow the user to interactively change values of variables, and see what are the effects of these changes on the other variables. Interestingly, this approach is orthogonal to the way in which variable values are depicted. This animated depiction gives an intuitive understanding of the way the variables relate to each other, and, thus, of the constraints placed on them. A static version of this representation can be built, as a 2-D grid in which the points which belong to the domains of two given variables are highlighted. This would show the combined effect of all the constraints on the two selected variables.

A hindrance to constraint debugging is the non-hierarchical, plain composition of the constraint store. Unlike programs and data, which are structured in many ways (procedures, predicates and rules, objects, tree structures, etc.), it is generally admitted that the store is a mere flat and huge collection of formulas with no structure whatsoever. Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (e.g., in Prolog IV). However, the programmer has usually in mind a structured vision of the program and the relationships of the elements in it. Understanding how the store relates to this vision is a very interesting task which we will look at more deeply below.

4.4 Abstracting

In executions of large programs, the debugging process has to cope with a sizeable number of calls, and with a large number of variables, having each many possible values, and related through many constraints. The programmer can be easily overwhelmed by the amount of information, so that no conclusion can be drawn directly from it. Thus, it is highly desirable to have methods which help to deal with these very important cases. Abstraction methods, amenable to be applied to as many cases as possible, would give the programmer a more friendly interface, by removing unneeded details.

In the case of control depiction as a search tree a clear possibility is to abstract parts of the tree, maybe by collapsing them [11]. The user might select which parts of the tree can be collapsed (perhaps those which correspond to parts of the program known to be correct), using either interactive graphical interface or assertions added to the program. It is also important not to lose important information when performing this abstraction: as an example, abstracting a tree by collapsing parts of it may cause data regarding the size of the tree not to be displayed. This data can alternatively be encoded as a tag in the collapsed tree, as a number or as a colour code. In general, many information items we want to retain, but which cannot be easily kept when abstracting, can be added with tags to the abstracted picture.

A similar case appears when displaying variables. In the particular case of finite domains, variables having large domains would need (if the aforementioned point-per-value representation is used) pictures with too many details in them. This would clutter the display, so that it would be difficult to draw any conclusion from the depiction. Also, quite often not all the values in the domain are of interest: if we are looking at how fast variables have their domains narrowed, it is usually of little interest which values are removed at each step. On the other hand, when we are more interested in knowing properties about the correctness of the domains of the variables, probably some points outside the current domain are not of interest (because we know they have been correctly removed), and some values currently in the domain are of interest only if they are removed (because we foresee they should be part of a final solution).

Some techniques, as domain compaction, which restricts which parts of a domain are to be visualised, can be devised and applied either by hand or sometimes in a semi-automatic fashion, maybe driven by annotations written by the user in the program source, or by pragmas automatically generated by analysers.

Too many variables to be visualised also call for abstraction of the display. When the domain visualisation tools are linked with tree visualisation tools, a natural option is to display only the variables in the node being studied. This is not always possible: the (very interesting) case of global constraints, in which many variables are related by a complex relationship, usually needs tailored visualisation, because it is not easy to reduce its solving process to a CLP-type search without implicitly dealing with the details of the solving algorithm (Chapter 12).

4.5 Controlling Size and Complexity of the Store (S-Boxes)

As mentioned above, no reasoning can be done on a flat representation of the store. This can be addressed by allowing the programmer to *structure the store by modifying the constraint granularity.* This would be achievable by structuring the store as a hierarchy of sub-stores, organised themselves in sub-stores etc. and giving the programmer the necessary tools to. Thus, the store is to be organised as an hierarchy of sub-stores. The tools supporting this idea should make it possible:

- to create and to modify the hierarchy,
- to focus on a local view of selected sub-stores,
- to navigate in two directions (search tree and propagation process) in these sub-stores.

Taking advantage of properties of the constraint narrowing operators and of the main algorithm computing the overall domain reductions, it is then possible to generate arbitrary levels of abstractions of the store by considering

subsets of the store as global constraints (in order to simplify the representation).

A tool for store inspection has been designed and is comprehensively described in Chapter 11. It allows the user to add structure information to the source code by *marking selected goals in selected rule bodies*. The set of "interesting" constraints (e.g., finite domain and/or interval constraints) defined by the corresponding predicates would then form a sub-store. Hierarchical organisation of the store is then associated to the program structure.

The key idea of the tool is to consider a store as a closed box (referred as *S-box* in the following) with no connection with the outside. By default, the store at any node of the search tree is considered as one S-box including all constraints of the store. This flat structure can be changed by using the tools to obtain an hierarchy of S-boxes, each of which includes a subset of the constraints involved. S-boxes can be created by highlighting some parts of the CLP program. This has the following effect, depending on what is highlighted:

- some constraints: they are all placed in one S-box;
- a goal: all constraints created when resolving the goal are included in one S-box;
- the head of a clause: all constraints created while in the clause are included in one S-box.

From the propagation view-point, a S-box acts as a big constraint which is the conjunction of all constraints it embeds. Note that this implies a modification of the propagation process: all constraints pertaining to the same S-box must be awaken in such a way that the whole propagation appears as atomic from the outside.

The set of constraints of the active S-box is displayed as a graph whose nodes are constraints and edges link constraints sharing variables. The graph contains only user constraints, that is decomposed constraints are reconstructed in order to be displayed. This requires modification in the core of the host system.

The hierarchy of existing S-boxes is represented in a tree form allowing the user to step backward and forward in the "zooming process".

The S-box approach can be seen as extracting constraints from the main CLP program, then gathering them in clauses to form a new structured Prolog program. The added structure facilitates debugging. In particular, it may allow more precise location of errors by the declarative debugger described in Chapter 5.

5 A Unified View of the Tools

This Section shows how the above mentioned tools relate to the proposed methodology and to each other. Figure 2 preserves the structure of Figure 1

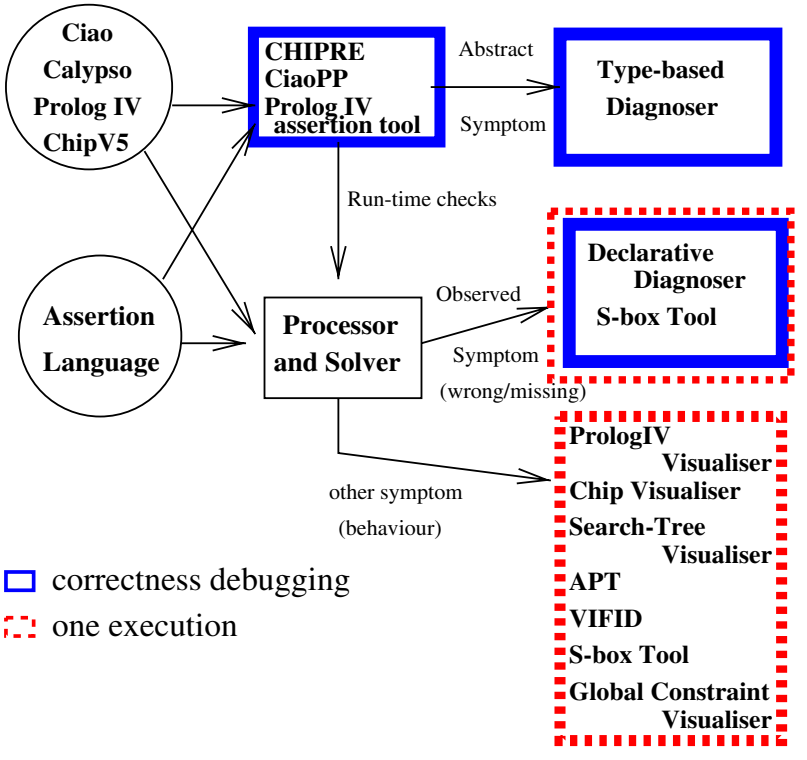


Fig. 2. DiSCiPl Debugging Methodology

to illustrate how the proposed methodology is supported by the tools. Most of the tools have been developed for selected individual platforms, as summarised below. However they follow the common view adopted in DiSCiPl.

As indicated in Figure 2, the platforms which have been used in the project are: Chip V5 [4], Prolog IV [10], Ciao [8], and `clp(fd)`/Calypso [6].

While the tools are platform-specific, the ideas represented by each of them are of general importance and can be integrated in any CLP platform. In some cases porting of the tools does not require a big effort. This was demonstrated by porting the type analyser used in the type diagnoser to Ciao, integrating it in CHIPRE, and by porting the type analyser and type diagnoser to Calypso. Also, the assertion language (see Chapter 1) has been used for different tools and on different platforms.

The rectangles with thick edges in the figure indicate tools for correctness debugging. They include the tools for static debugging:

– **CHIPRE** and **CiaoPP** (instances of the **Generic Preprocessor**) (Chapter 2) use abstract interpretation to infer assertions about the program at hand. The inferred assertions are compared with the specification (check

assertions), which may or may not be provided by the user, and with system assertions describing library predicates. If they do not conform the user is warned. For the check assertions that cannot be proved nor disproved run-time tests can be inserted in the program. Erroneous constructs are located by means of disproved check assertions and are reported to the user.
- The **Prolog IV assertion tool** (Chapter 3) verifies check assertions written in a restricted language, which must be provided a priori, and reports errors located by means of disproved assertions. Run-time tests can be included for assertions that cannot be proved nor disproved by the tool.
- The **Type-based Diagnoser** (for CHIP and for Calypso)(Chapter 4) uses the static analyser for inferring types. Diagnosis can be requested when one of the inferred types is different from that expected by the user (*abstract symptom*). Check assertions necessary for locating the error are requested interactively until an error message is obtained.

The rectangles with split edges indicate tools for debugging based on run-time symptoms.

The **Declarative Diagnoser** implemented for Calypso (Chapter 5) is an interactive tool that locates errors causing symptoms of wrong or missing answers.

A lot of effort has been devoted to performance analysis, since it is probably the most difficult aspect of constraint debugging. The tools for performance analysis are visualisation tools which facilitate understanding of a single execution of the debugged program. As discussed above, the problems of particular interest are: visualisation of the search space, display of variable bindings and constraints, and analysis of the behaviour of constraints.

As mentioned before, the whole search space during a single execution of a CLP program can be represented by a tree structure. Several equivalent presentations of this structure are possible. In CLP programs over finite domains the performance depends heavily on labelling. Therefore, it may be desirable to have a separate visualisation tool only for labelling. These considerations are behind the design decisions for the DiSCiPl visualisation tools for search space. The following tools have been developed for this purpose:

- **Prolog IV Search-Tree Visualiser** (Chapter 6) presents the search space as a three-dimensional dynamic tree based on the trace box model.
- **CHIP Labelling Visualiser**(Chapter 7) presents the labelling part of the search space of an execution of a finite domain CHIP program.
- **INRIA Search-Tree Visualiser** (Chapter 8) visualises the traversed search space (including labelling) as an SLD tree and includes abstraction functionalities. The visualiser has been developed for Calypso. A special focus is on view abstraction, so that the same search space can be represented in different ways. Properties defined with assertions are used for that purpose.

– The **APT** visualiser (Chapter 9) uses and/or trees for representing the search space. It has been developed for Ciao and was also ported to Calypso.

In all the above mentioned tools the nodes of the search space visualised give access to data stored in the corresponding state of the computation. However, this may not be sufficient to understand the behaviour of constraints during the computations. The latter problem is addressed by the following tools:

– **VIFID/TRIFID** (Chapter 10) is a tool to visualise the data evolution of CLP(FD) programs in Ciao. It depicts in an intuitive way the state of the constraint store at selected points in the program. It gives the user several facilities, including allowing to post/un-post arbitrary constraints during the execution of the program. It also shows the evolution of (a subset of) the variables in the program at different levels of abstraction.
– The **S-Box** tool (Chapter 11) for `clp(fd)` facilitates analysis of constraint propagation by allowing the user to impose some temporary structure on the constraint store. It may be used both for performance debugging and for correctness debugging as a support tool for the declarative diagnoser.
– **Global Constraint Visualiser** (Chapter 12) for CHIP makes it possible to visualise the global constraints `cumulative, diffn, cycle` and `among`. This is of great practical importance since the global constraints are heavily used in all industrial applications and due to their sophisticated nature it may be very difficult to understand their behaviour.

Some of the tools discussed in the book have been already used in industrial applications. Chapter 13 discusses this experience and draws some conclusions from it.

References

1 A. Aggoun, F. Benhamou, F. Bueno, M. Carro, P. Deransart, W. Drabent, G. Ferrand, F. Goualard, M. Hermenegildo, C. Lai, J. Lloyd, J. Małuszyński, G. Puebla, and A. Tessier. *CP Debugging Tools.* Public Deliverable D.WP1.1.M1.1, ESPRIT IV Project DiSCiPl, http://discipl.inria.fr/deliverables1.html, 1997.
2 N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1993.
3 L. Byrd. Understanding the control flow of prolog programs. In S.-A. Tärnlund, editor, *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.
4 Cosytec SA. *CHIP System Documentation*, 1998.
5 P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

6 D. Diaz. *GNU-Prolog user Manual.* http://pauillac.inria.fr/ diaz/gnu-prolog/, 2000
7 European Computer Research Center. *Eclipse User's Guide*, 1993.
8 M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming.* Nova Science, Commack, NY, USA, April 1999.
9 M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`
10 PrologIA. *Prolog IV Manual*, 1996.
11 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In Lee naish, editor, *ICLP'97.* MIT Press, July 1997.
12 E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1982.

1. An Assertion Language for Constraint Logic Programs

Germán Puebla, Francisco Bueno, and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, S-28660-Madrid, Spain
email: {german,bueno,herme}@fi.upm.es

In an advanced program development environment, such as that discussed in the introduction of this book, several tools may coexist which handle both the program and information on the program in different ways. Also, these tools may interact among themselves and with the user. Thus, the different tools and the user need some way to communicate. It is our design principle that such communication be performed in terms of *assertions*. Assertions are syntactic objects which allow expressing properties of programs. Several assertion languages have been used in the past in different contexts, mainly related to program debugging. In this chapter we propose a general language of assertions which is used in different tools for validation and debugging of constraint logic programs in the context of the DiSCiPl project. The assertion language proposed is parametric w.r.t. the particular constraint domain and properties of interest being used in each different tool. The language proposed is quite general in that it poses few restrictions on the kind of properties which may be expressed. We believe the assertion language we propose is of practical relevance and appropriate for the different uses required in the tools considered.

1.1 Introduction

Assertions are linguistic constructions which allow expressing properties of programs. Assertions have been used in the past in different contexts and for different purposes related to program development:

Run-time checking: This is one of the traditional uses of assertions and has been applied extensively in the context of imperative programming languages. The user adds assertions to a program which express conditions about the program which should hold at run-time. Otherwise the program is *incorrect*. A usual example is to check that the value of a variable remains within a given range at a given program point. If assertions are found not to hold an error message is flagged. Note that in this context, assertions express properties about the run-time behaviour of the program which *should hold* if the program is correct (see [1.30] for an application in Constraint Logic Programming (CLP) [1.19]).

Replacing the oracle: In declarative debugging [1.27], the existence of an *oracle* (normally the user) which is capable of answering questions about

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 23–61, 2000.

the intended behaviour of the program is assumed. In this context, the user may write assertions which express properties which should hold if the program were correct [1.11, 1.12, 1.2]. If it is possible to answer the questions posed by the declarative debugger just by using the information given as assertions, then there is no need to ask the oracle (the user). This makes systems more practical because the burden on the user is reduced. Note that here again, assertions are used to express properties which *should hold* for the program to be correct.

Compile-time checking: In this context, the user may write assertions which express properties about the program which are intended to be checked at compile-time. The result of such checking may indicate either that the assertions actually hold and the program is *validated* w.r.t. the assertions or that the assertions do not hold, and then the program is *incorrect* w.r.t. the assertions. Again, these are properties which *should hold*, i.e., otherwise a bug exists in the program. An example of this kind of assertions are type declarations (e.g., [1.18, 1.28], functional languages, etc.), which have been shown to be useful in debugging. Generally, and in order to be able to check these properties at compile-time, the expressible properties are restricted in such a way that compile-time checking can always determine whether the assertions hold or not.

Providing information to the optimiser: Assertions have also been proposed as a means of providing information to an optimiser in order to perform additional optimisations during code generation. Such assertions can be provided by the user (e.g., [1.28], which also implements checking) or automatically generated, generally by means of static program analysis. In this context, assertions do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

General communication with the compiler: In a setting where there is both a static inference system, such as an abstract interpreter [1.9, 1.14], and an optimiser, assertions have also been proposed as a means of allowing the user to provide additional information to the analyser [1.4], which it can use both to increase the precision of the information it infers and/or to perform additional optimisations during code generation [1.31, 1.29, 1.22, 1.20]. Examples are assertions which state information on *entry points* to a program module, assertions which describe properties of built-ins [1.17], assertions which provide some (but not all [1.4]) type declarations in the context of a type *inferencing* system, etc. Also, assertions can be used to represent analysis output in a user-friendly way and to communicate different modules of the compiler which deal with analysis information [1.4]. In this context, assertions express both properties which should hold and properties which *do hold* for the program in hand.

Program documentation: Assertions have also been used to document programs and to automatically generate manuals (as inspired by the "literate programming" style [1.21, 1.7]). These assertions are usually written by the user but they can also be automatically generated. Some examples are Javadoc [1.13], in the context of imperative languages, and LPdoc in the context of CLP [1.15]. In this application, assertions may express both properties which *do hold* or which *should hold* for the program in hand.

In addition to the classification given above, made according to the context in which assertions are used, assertions can be classified according to many other criteria. For example, as mentioned above, in some cases the assertions express properties which should hold (intended properties) while in others the assertions express properties which actually hold (actual properties) for the program. Also, it can be noted that in some cases it is the user who provides assertions to the tool whereas in other ones the assertions are generated by the tool.

Independently of these and other classifications, our aim is to design an assertion language which, in the context of CLP, *can be used for all the purposes mentioned above*, and hopefully new ones which may result from synergistic interactions due to the integration. The main application in which we will be using the assertion language within this book will be, of course, program debugging, but we expect the language to be useful in several other tasks.

There is a clear trade-off between the expressive power of a language of assertions and the difficulty in dealing with it. A reasonable overall objective when designing an assertion language is to try to maximise the expressive power of the language while at the same time keeping it amenable to automatic reasoning. More concretely, in the context of the DiSCiPl project, different tools for program development and debugging co-exist in the programming environment. In particular, Chapter 2 presents a preprocessor which performs combined compile-time and run-time checking of assertions, inference of assertions based on abstract interpretation, a form of diagnosis (location of errors), and, though not discussed there, also automatic documentation generation from assertions. The system presented in Chapter 3 also performs compile-time and run-time checking of assertions. Chapter 4 presents a system which allows locating errors in programs and uses assertions restricted to regular types. Finally, assertions could be used to replace the oracle in the declarative debugger presented in Chapter 5 (and even have some potential uses in the context of the visualisers described in later chapters).

We would like the assertion language to allow expressing any property which is of interest for any of the debugging (and validation) tools in the environment. Also, we would like the assertion language to be independent of the particular CLP platform in which it is applied and the constraint domains supported. Thus, we choose not to restrict too much beforehand the

kind of properties which can be expressed with our assertions. A fundamental motivation behind this choice is the frequent availability in our target debugging environments of tools which can handle quite rich properties, through techniques such as approximations and abstract interpretation [1.9].

Clearly, not all tools will be capable of dealing with *all* properties expressible in our assertion language. However, rather than having different assertion languages for each tool, we propose the use of the same assertion language for all of them. This facilitates communication among the different tools and enables easy reuse of information, i.e., once a property has been stated there is no need to repeat it for the different tools. Each tool should then only make use of the part of the information given as assertions which the tool *understands* and should deal *safely* with the part of the information it does not understand.

Informally, a particular tool understands a given assertion if the tool can evaluate the assertion in the appropriate context and this evaluation has a chance of yielding true or false (i.e., it is not the case that it will always return "don't know"). For example, a program analysis for groundness of the computed answers typically understands an assertion stating that in all answers to a given predicate the second argument is ground. It also may understand an assertion stating that the same argument is a free variable (in the sense that it may be able to prove that the assertion is false). However, it will not understand an assertion which states that all calls to a given predicate must have a list as first argument: first, the tool is not able to reason about calls to predicates; second, it is not able to reason about "types."

We will present assertions which are able to capture "contexts" of the operational semantics as well as of the declarative semantics of CLP programs. Properties about the program execution states, of the computed answers, the correct answers, and of the computations themselves can all be expressed in our assertions. A preliminary version of the assertion language we present here appeared in [1.25].

In our assertion language, assertions are always instances of some *assertion schema* together with a reference to which part of the program (predicate or program point) the assertion refers to and, depending on the schema used, one or two *logic formulae*. Whereas the assertion language has a fixed set of assertion schemas, the user has a high degree of freedom for defining the logic formulae for the properties considered of interest. Thus, the whole assertion language is determined by a set of assertion schemas and the way in which "logic formulae" can be built. Intuitively, the logic formulae in the assertions are used to say things such as "X is a list of integers," "Y is ground," "$p(X)$ does not fail," etc. The (schemas of the) assertions specify which are the X's, Y's, and $p(X)$'s of which the previous things are said.

The structure of this chapter is the following. The role of assertions in program validation and debugging is further discussed in Section 1.2. Sections 1.3 through 1.6 present several assertion schemas available in our lan-

guage. In more detail, Section 1.3 presents a series of assertion schemas which allow expressing properties related to execution states. Section 1.4 presents the syntax we use for logic formulae and also discusses some general issues on the evaluation of such formulae. While the assertion schemas presented in Section 1.3 allow expressing properties related to execution states and are thus operational, Section 1.5 introduces an assertion schema related to declarative properties of programs. In Section 1.6 we present assertion schemas which allow reasoning about completeness of the set of answers of a program. Section 1.7 shows how, independently of the schema used, the assertion language is made more expressive by adding a flag to each assertion which we refer to as an assertion "status." Section 1.8 presents yet another assertion schema which is conceptually different from all the ones seen in the previous sections. It allows expressing properties about whole computations of predicates rather than just states. In Section 1.9 we present how to define "property predicates", i.e., the predicates which are used as atomic formulae. These predicates are the building blocks from which logic formulae, and thus assertions, are written. Section 1.10 summarises the assertion syntax and introduces some extensions to the assertion language which make the task of assertion writing easier. Finally, Section 1.11 discusses some issues about the proposed assertion language and concludes.

1.2 Assertions in Program Validation and Debugging

When reasoning about whether a certain program behaves as indicated by a set of assertions, it is often useful to restrict the discussion to a set of *valid* initial queries. This is because when we design a program not only do we have an expectation of what the program must compute but also we expect the program to be used by calling only certain predicates, and with some restricted class of input data. Thus, informally, a program is correct when it behaves according to the user's intention for any input data satisfying certain preconditions. We refer to such input data as *valid* input data, and to the corresponding queries as *valid queries*. The `entry` assertions which will be presented in Section 1.3.4 are a means for providing (a description of) the valid queries to the program. In what follows we assume that program debugging and validation is always performed w.r.t. a given set of (descriptions of) valid queries.

Assertion-based program validation and debugging aims at *automatically* reasoning about program correctness by checking whether assertions hold or not for a given program and a (set of) valid initial queries. In order to perform such reasoning automatically, some *inference system* is required which is capable of determining whether the assertions hold or not. Most existing assertion-based systems are designed with a fixed inference system in mind (for example, a particular type inferencing algorithm). Depending on the capabilities of such inference system, the assertion language is defined in

such a way that every assertion expressible in the assertion language can be automatically determined to hold or not. In the design of the present assertion language we depart from most existing assertion languages in that we do not assume the existence of any fixed inference system. The main reason for this is the availability of a growing number of practical different static analyses for (constraint) logic programs which can perform the task of inference systems. We admit the possibility that different tools use different inference systems and, also, the same tool may make use of several inference systems. Thus, we cannot assume that the given assertions can always be proved nor disproved by any particular inference system. This on one hand allows having a very flexible assertion language since it is not limited to some kinds of properties. On the other hand, each tool must know how to safely deal with those assertions for which its inference system(s) cannot determine whether they hold or not. Though plenty of different inference systems may exist, we make a distinction between static inference systems and dynamic inference systems. The former systems are capable of reasoning about the program behaviour (and thus of the truth value of the assertions) without actually having to run the program, whereas the latter systems do run the program and check the assertions which are triggered by the execution of the program. In the design of the assertion language we have taken both kinds of inference systems into account, providing means to deal with the program properties expressed by the assertions either in dynamic or in static systems. We postpone the discussion on how this can be done, and how to safely deal with properties which a particular inference system does not understand until after (part of) the assertion language has been presented.

Very often, the properties of a program which we are interested in expressing by means of assertions are related to the run-time behaviour of the program. For this, we need to consider the *operational* semantics of the program. The operational semantics of a program is in terms of its *derivations* which are sequences of reductions between *execution states*. An execution state $\langle G \mid \theta \rangle$ consists of the current goal G and the current constraint store (or *store* for short) θ which contains information on the values of variables. The way in which a state is transformed into another one is determined by the operational semantics and the program code. The assertions presented in sections 1.3 and 1.8 refer to the operational behaviour of the program.

One of the advantages of CLP is that in addition to the operational semantics, programs also have a *declarative* meaning or semantics which is independent of the particular details on how the program is executed. Our assertion language also has assertions specifically designed for expressing properties related to such declarative semantics. These assertions are presented in Section 1.5.

Every assertion A is conceptually composed of two logic formulae which we refer to as app_A and sat_A. Evaluation of these logic formulae should return either the value *true* or the value *false* when evaluated on the corresponding

context (i.e., execution state, correct answer, computation, or whatever is the "semantic context" which the assertion refers to) by using an appropriate inference system. The formula app_A determines the *applicability set* of the assertion: a context s is in the applicability set of A iff app_A takes the value *true* in s. Also, we say that an assertion A is *applicable* in context s iff app_A holds in s. The formula sat_A determines the *satisfiability set* of the assertion: a context s is in the satisfiability set of A iff sat_A takes the value *true* in s. If we can prove that there is a context which is in the applicability set of an assertion A but is not in its satisfiability set then the program is definitely incorrect w.r.t. A. Conversely, if we can prove that every context in which A is applicable is in the satisfiability set of A then the program is validated w.r.t. A.

In this chapter we will present a repertoire of *assertion schemas*. Such schemas can be seen as templates which when properly instantiated define in a simple and clear way the required formulae app_A and sat_A. The choice of one schema or another greatly determines what the applicability contexts of the assertion may be. Thus, the use of assertion schemas on one hand makes the task of assertion writing easier, but on the other hand somewhat limits the freedom in describing in which contexts assertions are applicable. However, we argue that the proposed repertoire of schemas is flexible enough for the purposes for which the assertion language has been designed and we accept this limited freedom in return for the clarity of the resulting assertion language.[1]

1.3 Assertion Schemas for Execution States

When considering the operational behaviour of a program, it is natural to associate (sets of) execution states with certain syntactic elements of the program. Typically, a program can be seen as composed of a set of *predicates* (also known as *procedures*). Alternatively, a program can be seen, at a finer-grained level, as composed of a set of *program points*. Thus, we first introduce several assertion schemas whose applicability context is related to a given predicate. Then we introduce an assertion schema whose applicability context is related to a particular program point. We refer to the former kind of assertions as *predicate* assertions, and to the second ones as *program-point* assertions. Though a simple program transformation technique can be used to express program-point assertions in terms of predicate assertions, we maintain program-point assertions in our language for pragmatic reasons.

As a general rule, we restrict the properties expressible by means of assertions about execution states to those which refer to the values of certain variables in the store of the corresponding execution state. This has the advantage that in order to check whether the app_A and sat_A logic formulae hold

[1] A formal discussion on applicability contexts and schemas can be found in [1.26].

or not it suffices to inspect the store at the corresponding execution state. Also, the variables (arguments) on whose value we may state properties are also restricted in some way. In the case of predicate assertions, the arguments whose value we can inspect are those in the head of the predicate. In the case of program-point assertions, they are the variables in the clause to which the program point belongs.

Example 1.3.1. We illustrate the use of assertions about execution states with an example. Figure 1.1 presents a Ciao [1.3] program which implements the *quicksort* algorithm together with a series of both predicate and program-point assertions which express properties which the user expects to hold for the program.[2] Two predicate assertions are given for `qsort/2` (`A1` and `A2`) and another two for `partition/4` (`A4` and `A5`). There is also a program-point assertion (`A3`). The meaning of the assertions in this example is explained in detail in subsequent sections. Note that more than one predicate assertion may be given for the same predicate. In such a case, all of them should hold for the program to be correct and composition of predicate assertions should be interpreted as their conjunction.

1.3.1 An Assertion Schema for Success States

This assertion schema is used in order to express properties which should hold on termination of any successful computation of a given predicate. They account for probably one of the most common sorts of program properties which we may be interested in expressing in relation with program predicates. They are similar in nature to the *postconditions* used in program verification. They can be expressed in our assertion language using the assertion schema:

`:- success` *Pred* `=>` *Postcond*.

This assertion schema has to be instantiated with suitable values for *Pred* and *Postcond*. *Pred* is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and *Postcond* is a logic formula about execution states (to be discussed in Section 1.4), and which plays the role of the sat_A formula. The resulting assertion should be interpreted as "in any activation of *Pred* which succeeds, *Postcond* should hold in the success state." Referring to our notions of applicability and satisfiability sets, the resulting assertion can be interpreted as "the assertion is applicable in those execution states which correspond to success states of a computation of *Pred*, and its satisfiability set has those states in which *Postcond* holds."

[2] Both for convenience, i.e., so that the assertions concerning a predicate appear near its definition in the program text, and for historical reasons, i.e., mode declarations in Prolog or entry and trust declarations in PLAI [1.4, 1.23], we write predicate assertions as directives, which appear within the program text. Depending on the tool different alternatives may be used, including for example separate files or incremental addition of assertions in an interactive graphical environment.

```
:- calls qsort(L,R) : list(L).                                    % A1
:- success qsort(L,R) : list(L,int) => list(R,int).               % A2

qsort([X|L],R) :-
        check(number(X)),                                         % A3
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

:- calls partition(L,X,L1,L2) : list(L).                          % A4
:- success partition(L,X,L1,L2)
         : ( list(L), ground(X) ) => ( list(L1), list(L2) ). % A5

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,
        partition(R,C,Left,Right1).
```

Fig. 1.1. Some Assertions about Execution States

Example 1.3.2. We can use the following assertion in order to require that the output (second argument) of procedure `qsort` for sorting lists be indeed sorted:

```
:- success qsort(L,R) => sorted(R).
```

Clearly, we are assuming that `sorted(R)` is interpreted in a suitable inference system, in which it takes the value true iff `R` is bound to a sorted list. The assertion establishes that this (atomic) formula is applicable at all execution states which correspond to a success of `qsort`.

An important thing to note is that in contrast to other programming paradigms, in (C)LP a call to a predicate may generate zero (if the call fails), one, or several success states, in addition to looping (or returning error). The postcondition stated in a `success` assertion refers to *all* the success states (possibly none).

1.3.2 Adding Preconditions to the Success Schema

The `success` schema can be used when the applicability set of an assertion is the set of success states for a given predicate. However, it is often useful to consider more restricted applicability sets. A classical example is to only consider those successful states which correspond to activations of the predicate

which at the time of calling the predicate satisfy certain *precondition*. The preconditions we consider are, in the same way as *Postcond*, logic formulae about states. The success schema with precondition takes the form:

`:- success` *Pred* `:` *Precond* `=>` *Postcond*`.`

and it should be interpreted as "in any invocation of *Pred* if *Precond* holds in the calling state and the computation succeeds, then *Postcond* should also hold in the success state." Alternatively, it can be interpreted as "the assertion is applicable to those execution states which correspond to success states of a computation of *Pred* which was originated by a calling state in which *Precond* holds, and its satisfiability set has those states in which *Postcond* holds." Note that '`:- success` *Pred* `=>` *Postcond*' is equivalent to '`:- success` *Pred* `:` *true* `=>` *Postcond*'.

It is important to also note that even though both *Precond* and *Postcond* are logic formulae about execution states, they refer to different execution states. *Precond* must be evaluated w.r.t. the store at the calling state to the predicate, whereas *Postcond* must be evaluated w.r.t. the store at the success state of the predicate.

Example 1.3.3. The following assertion (`A2` in Figure 1.1) requires that if `qsort` is called with a list of integers in the first argument position and the call succeeds, then on success the second argument position should also be a list of integers:

```
:- success qsort(L,R) : list(L,int) => list(R,int).
```

where `list(A,int)` is an atomic formula which takes the value true iff `A` is bound to a list of integers in the corresponding state. Note that the program in Figure 1.1 can be used to sort a list of integers but also to sort a list of, say, floating point numbers. Thus, we cannot require in general that the result of ordering a list be a list of integers. This is why we add as precondition that the list to be sorted is indeed a list of integers.

1.3.3 An Assertion Schema for Call States

We now introduce an assertion schema whose aim is to express properties which should hold in any call to a given predicate. These properties are similar in nature to the classical *preconditions* used in program verification. A typical situation in which this kind of assertions are of interest is when the implementation of a predicate assumes certain restrictions on the values of the input arguments to the predicate. Such implementation is often not guaranteed to produce correct results unless such restrictions hold. Assertions built using this schema can be used to check whether any of the calls for the predicate is not in the expected set of calls (i.e., the call is "inadmissible" [1.24]). This schema has the form:

`:- calls` *Pred* `:` *Precond*`.`

This assertion schema has to be instantiated with a predicate descriptor *Pred* and a logic formula about execution states *Precond*. The resulting assertion should be interpreted as "in all activations of *Pred* the formula *Precond* should hold in the calling state." Alternatively, the resulting assertion can be interpreted as "the assertion is applicable in those execution states which correspond to calling states to *Pred*, and its satisfiability set has those states in which *Precond* holds."

Example 1.3.4. The following assertion (`A1` in Figure 1.1) built using the `calls` schema expresses that in all calls to predicate `qsort` the first argument should be bound to a list:

```
:- calls qsort(L,R) : list(L).
```

1.3.4 An Assertion Schema for Query States

It is often the case that one wants to describe the exported uses of a given predicate, i.e., its valid queries. This is for example the case also in traditional preconditions of a program. Thus, in addition to describing calling and success states, we also consider using assertions to describe *query states*, i.e., valid input data. In terms of the operational semantics, in which program executions are sequences of states, query states are the initial states in such sequences. These can be described in our assertion language using the `entry` schema, which has the form:

`:- entry` *Pred* `:` *Precond*.

where, as usual, *Pred* is a predicate descriptor and *Precond* is a logic formula about execution states. It should be interpreted as "*Precond* should hold in all initial queries to *Pred*." Alternatively, it can be interpreted as "the assertion is applicable in those execution states which correspond to initial queries to *Pred*, and the satisfiability set has those states in which *Precond* holds."

Example 1.3.5. The following assertion indicates that the predicate `qsort/2` can be subject to top-level queries provided that such queries have a list of numbers in the first argument position:

```
:- entry qsort(L,R) : numlist(L).
```

The set of all `entry` assertions is considered *closed* in the sense that they must cover all valid initial queries. This is equivalent to considering that an assertion of the form '`:- entry` *Pred* `: false.`' exists for all predicates *Pred* for which no `entry` assertion has been provided.

It can be noted that `entry` and `calls` schemas are syntactically (and semantically) similar. However, their applicability set is different. The assertion in the example above only applies to the initial calls to `qsort`, whereas, for example, the assertion '`:- calls qsort(L,R) : numlist(L).`' applies to any call to `qsort`, including all recursive (internal) calls. Thus, `entry` assertions

allow providing more precise descriptions of initial calls, as the properties expressed do not need to hold for the internal calls.

Example 1.3.6. Consider the following program with an entry assertion:

```
:- entry p(A) : ground(A).
p(a).
p(X):- p(Y).
```

If instead of the `entry` above we had written '`:- calls p(A) : ground(A).`' then such assertion would not hold in the given program. For example, the execution of `p(b)` produces calls to `p` with the argument being a free variable. However, the execution of `p(b)` satisfies the `entry` assertion since the internal calls to `p` are not in the applicability context of the assertion.

The name `entry` is used for historic reasons. Entry declarations have long been used (see for example [1.4]) in order to improve the accuracy of goal-dependent analyses since they allow providing a description of the *initial* calls to the program. Goal-dependent analyses may obtain results which are *specialized* (restricted) to a given context, which allows them to provide in general better (stronger) results than goal-independent analyses.

1.3.5 Program-Point Assertions

As already mentioned, usually, when considering operational semantics of a program, in addition to predicates we also have the notion of *program points*. The program points that we will consider are the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. For simplicity, we add program-point assertions to a program by adding a new literal at the corresponding program point. This literal is of the form:

`check`(*Cond*).

an it should be interpreted as "whenever execution reaches a state originated at the program point in which the assertion is, *Cond* should hold." Intuitively, each execution state can be seen as originated at a given program point. Thus, alternatively it can be interpreted as "the assertion is applicable in those execution states originated at the program point in which the assertion appears and its satisfiability set has those states in which *Cond* holds."

Example 1.3.7. Consider the following clause '`p(X):- q(X,Y), r(Y).`' Imagine for example that whenever the clause is reached by execution, after the successful execution of the literal `q(X,Y)`, `X` should be greater than `Y` and `Y` should be positive. This can be expressed by replacing the previous clause by the following one in which a program-point assertion has been added:

```
p(X):-  q(X,Y), check((X>Y,Y>=0)), r(Y).
```

An important difference between program-point assertions and predicate assertions is that while the latter are not part of the program, program-point assertions are, as they have been introduced as new literals in some program clauses. In order to avoid program-point assertions from changing the behaviour of the program (at least if dynamic checking has not been enabled), we assume that the predicate `check/1` is defined as

```
check(_Prop).
```

i.e., any call to `check` trivially succeeds. If dynamic checking is being performed, this definition is overridden by another one which actually performs the checking. One possible such definition for run-time checking is presented in Chapter 2.

1.4 Logic Formulae about Execution States

As we have seen, schemas for predicate assertions have to be instantiated with a predicate descriptor *Pred* and one or two logic formulae on execution states, and the schema for program-point assertions also has to be instantiated with a logic formula about execution states. In this section we present how such formulae are defined in our assertion language, and discuss how they should be evaluated.

We allow conjunctions and disjunctions in the formulae, and choose to write them down, for simplicity, in the usual CLP syntax. Thus, logic formulae about execution states can be:

- An atom of the form $p(t_1, \ldots, t_n)$ with $n \geqslant 0$, where p/n is a *property predicate*. How to define these predicates is explained in Section 1.9.
- An expression of the form ($F1$, $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the comma should be interpreted as conjunction.
- An expression of the form ($F1$; $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the semicolon should be interpreted as disjunction.

Such formulae have to be evaluated as part of the evaluation of an assertion. Evaluation of an assertion can be seen as composed of three steps. First, an appropriate inference system[3] IS must be used to evaluate each of the atomic formulae AF of the assertion on the appropriate store θ. This presents a technical difficulty in the case of predicate assertions, since the formula is referred to the variables of the predicate descriptor $Pred$ in the assertion, whereas it has to be evaluated on a store θ which refers to variables different from those in *Pred*. We assume that a consistent renaming

[3] As we will see in Chapter 2, even the underlying logic system may be used as inference system for evaluating the formulae.

has been applied on the assertion, and thus on AF, so that it refers to the corresponding variables of θ. We denote by $eval(AF, \theta, P, IS)$ the result of the evaluation of AF in θ by IS w.r.t. the definitions of property predicates P (the definition of property predicates in discussed in Section 1.9). The inference system must be correct in the sense that if $eval(AF, \theta, P, IS) = true$ then AF must actually hold in θ and if $eval(AF, \theta, P, IS) = false$ then AF must actually do not hold in θ. However, we also allow incompleteness of IS, i.e., $eval(AF, \theta, P, IS)$ does not necessarily return either *true* or *false*. If IS is not able to guarantee that AF holds nor that it does not hold in θ then it can return AF itself. Thus, if $eval(AF, \theta, P, IS) = AF$ it can be interpreted as a "don't know" result.

The second step involves obtaining the truth value of the logic formulae app_A and sat_A as a whole from the results of the evaluation of each atomic formula. For this, standard simplification techniques for boolean expressions can be used. We denote by $simp(F)$ the result of simplifying a logic formula F. Since $eval(AF, \theta, P, IS)$ may take the value AF for some atomic formulae in F, $simp(F)$ may take values different from *true* and *false*, which are not simplified further.

The third step corresponds to obtaining the truth value of the assertion as a whole from the values obtained for $simp(app_A)$ and $simp(sat_A)$. The assertion is proved to hold either if $simp(app_A) = false$ or $simp(sat_A) = true$. The assertion is proved not to hold if $simp(app_A) = true$ and $simp(sat_A) = false$. Once again, we may not be able to prove not to disprove the assertion if $simp(app_A)$ and/or $simp(sat_A)$ are not either *true* nor *false*. A program is correct for given valid queries if all its assertions have been proved for all the states that may appear in the computation of the program with the given queries (see [1.26] for a formal presentation of correctness and completeness w.r.t. these kinds of assertions).

In order to compute the value of $eval(p(t_1, \ldots, t_n), \theta, P, IS)$ three cases are considered. The first one is that IS is *complete* w.r.t. p/n, i.e., it can always return either *true* or $false$ for any store θ and any terms $t_1, \ldots, t_n$. The second case is when IS can return the value *true* or the value $false$ for some store θ and terms $t_1, \ldots, t_n$ but not for all. In this case we say that IS *partially captures* the predicate p/n. This is usually based on sufficient conditions. The third case is when IS cannot return the value *true* nor the value $false$ for any θ, i.e., IS *does not capture* (or it does not "understand") p/n.

Usually, given an inference system IS, there is a set of property predicates for which IS is *complete*. In addition, the user can often define other predicates for which IS is also complete by using some fixed and restricted syntax (consider, for example, defining a new type). The assertion language has to provide means to do this. Similarly, we call a predicate *provable* (resp. *disprovable*) in IS if IS can sometimes evaluate it to *true* (resp. $false$). Since from the beginning we allow incompleteness of IS, it is important that the

user can indicate property predicates which are *provable* and *disprovable* in a given inference system, together with the corresponding sufficient conditions. Our assertion language also provides means to do this, as explained in Section 1.9.

The previous discussion assumes that the store on which the logic formulae are evaluated is given. This is feasible when assertions, and thus logic formulae, are evaluated at run-time, since the store θ is available. However, if static checking is being performed, only descriptions of stores and execution states rather than exact knowlegde on such stores is available. There are two reasons for this. One is that at compile-time the actual values of the (valid) input data to the program are usually not available. The second one is that in order to ensure termination of static checking, some approximation of the actual computation must be performed which loses part of the information on the actual execution states.

In return for the loss of information introduced by static checking, static analysis systems often compute safe approximations of the stores reached during computation. This makes it possible to validate the program w.r.t. the assertions [1.5], since the results of analysis include all valid executions of the program. Thus, if a property can be proved in a safe approximation of a store θ then it is also proved to hold in θ; if it can be proved that it does not hold in the approximation of θ then it does not hold either in θ. This is done for example in the preprocessor presented in Chapter 2, using abstract interpretation to compute the approximations.

1.5 An Assertion Schema for Declarative Semantics

As already mentioned, one of the main features of CLP is the existence of a *declarative semantics* which allows concentrating on *what* the program computes and not on *how* it should be computed. This feature is exploited for example in declarative debugging. Those tools which are concerned with the declarative semantics require the use of dedicated assertions. This is why we also have in our assertion language assertion schemas which refer to declarative semantics.

Consider the case of CLP(D), where D is the domain of values. For example, in classical logic programming D is the Herbrand Universe. In CLP($\Re$), D is the set of real numbers and of ground terms (for example lists) containing real numbers. The declarative semantics in CLP(D) associates a *meaning* to each program P, denoted $[\![P]\!]$, which corresponds to the *least D-model* of P. $[\![P]\!]$ is a set of D-atoms, where a D-atom is an expression $p(d_1, \ldots, d_n)$ with $n \geqslant 0$ such that p is an n-ary predicate symbol and $d_i \in D$. $[\![P]\!]$ coincides with $\mathrm{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\varnothing)$ where T_P is the immediate consequence operator.

We now introduce an assertion schema which allows stating properties which should hold in the least D-model $[\![P]\!]$ of a program. Otherwise the program is incorrect. This can be done using the schema:

`:- inmodel` *Pred* `=>` *Cond*.

where *Pred* is a predicate descriptor and *Cond* is a logical formula about D-atoms. It should be interpreted as "in any D-atom $p(d_1, \ldots, d_n) \in [\![P]\!]$ whose predicate symbol coincides with that of *Pred*, *Cond* should hold." Alternatively, it can be interpreted as "the applicability set of the assertion has those D-atoms in $[\![P]\!]$ whose predicate symbol is that of *Pred* and the satisfiability set is the set of D-atoms whose predicate symbol is that of *Pred* which satisfy the property *Cond*."[4]

Example 1.5.1. The following assertion states that the result of ordering a list by means of the predicate `qsort` should be a list:

```
:- inmodel qsort(L,R) => list(R).
```

if we determine that the D-atom `qsort(a,a)` $\in [\![P]\!]$ then the program is not correct w.r.t. the above assertion A. This is because in `qsort(a,a)` A is applicable. However, $list(a)$ does not hold and thus `qsort(a,a)` is not in the satisfiability set of A.

As seen in the example above, this kind of assertions allows reasoning about (partial) correctness of programs. This is why we call them *correctness* declarative assertions. Note that they are apparently very similar to the `success` assertions presented in Section 1.3.1 since every success state of a predicate is in the declarative semantics of the program. In fact, a program which is correct w.r.t. an assertion '`:- inmodel` *Pred* `=>` *Cond*' is also correct w.r.t. the assertion '`:- success` *Pred* `=>` *Cond*' due to correctness of the operational semantics (but not vice versa due to possible incompleteness of the operational semantics). A further difference between `inmodel` and `success` assertions is that in `inmodel` it is not possible to add preconditions since the declarative semantics does not capture calls to predicates. In addition, depending on the semantics used, the logic formula used in *Cond* of `inmodel` assertions are not allowed to refer to the instantiation state of arguments, for example using `var/1`, whereas this is completely legal in `success` assertions.[5]

1.6 Assertion Schemas for Program Completeness

As seen above, there is a similarity between `success` and `inmodel` assertions in that they both express properties about the *answers* of predicates.

[4] Any D-atom p with the predicate symbol of *Pred* satisfies $p = \mu(Pred)$ for a substitution μ. Strictly speaking, it is $\mu(Cond)$ which is satisfied on p. For simplicity of the presentation, we have preferred not to clatter it with these technical details.

[5] However, this could be remedied by using a more powerful declarative semantics (e.g. [1.1]), in which for example using `var/1` would make perfect sense.

More precisely, `success` assertions express properties of the *computed answers* of predicates, i.e., those generated by the operational semantics, whereas `inmodel` assertions refer to *correct answers*, i.e., those which are in the declarative semantics of the program (its least D-model). When considering answers to predicates, one particular aspect to reason about is *correctness* of the program, which corresponds to answering the question: Are all the actual answers of the program in the set of intended answers? Conversely, another aspect we can reason about is the well known concept of *completeness* of the program, which corresponds to answering the question: Are all the intended answers of the program in the set of actual answers? In other words, a program is complete when it does not fail to produce any expected answer. Clearly, we would like our program to be both correct and complete w.r.t. our intention. This corresponds to the classical notion of *total correctness*, as opposed to the previous notion of correctness, which is also known as *partial correctness*.

Though not explicitly mentioned, all the assertions presented in the previous sections allow reasoning about (partial) correctness of programs w.r.t. assertions, i.e. the may allow detecting that a program is not partially correct w.r.t. the assertion or validating the program w.r.t. the assertion. However, they are of no use in order to reason about completeness of programs. This is why we now introduce another kind of assertions which are variations of the `inmodel` and `success` assertions. They can be distinguished from the previous ones because the arrow (`=>`) now points in the reverse direction, i.e., `<=`. For example, an assertion of the form:

`:- inmodel` *Pred* `<=` *Cond*`.`

(note the reversed direction in the arrow) should be interpreted as "any D-atom of the form $p(d_1, \ldots, d_n)$ whose predicate symbol is the same as that in *Pred* and on which *Cond* holds should be in $[\![P]\!]$."

Example 1.6.1. The following assertion where the symbol `==` stands for term identity states that the pair $\langle$`[2,1]`,`[1,2]`$\rangle$ is an expected solution of `qsort`.:

```
:- inmodel qsort(L,R) <= (L == [2,1], R == [1,2]).
```

If we can determine that $qsort([2,1],[1,2]) \notin [\![P]\!]$ then P is incomplete w.r.t the above assertion. This is an indication that the existing code for `qsort` does not allow determining that one (in this case the only) result of ordering `[2,1]` is `[1,2]`. It is thus an indicator that the current version of the program is not complete w.r.t. the above assertion. This is clearly the symptom of an error, but it can be the case that the program sorts correctly lists of length different from two, in which case the error cannot be detected automatically using the (partial correctness) assertions seen in the previous sections.

We can also write completeness assertions for operational semantics using the following schema (optional "fields" appear in square brackets):

`:- success` *Pred* [`:` *Precond*] `<=` *Postcond*.

which should be interpreted as "any call to predicate *Pred* which on the calling state satisfies *Precond* must have as success states at least all those states which satisfy *Postcond*."

Example 1.6.2. Consider the following program which aims at improving the previous version of `qsort` by stopping recursion when the list has length 1 since any such list is already sorted. We add to the program a completeness success assertion:

```
:- success qsort(L,R): (L==[], var(R)) <= R == [].
qsort([X,Y|L],R) :-
        partition([Y|L],X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([A],[A]).
```

which requires that the results of sorting the empty list include the empty list, provided that the second argument satisfies `var/1` which holds iff it is a free, unconstrained variable at the call. The precondition `var(X)` is needed since, for example, the call '`qsort([],[1])`' has no success state.

The assertion above can be used to detect that the program is not complete since we have forgotten the clause '`qsort([],[]).`', thus a call such as '`qsort([],L).`' fails without producing any answer.

1.7 Status of Assertions

Assertions can be used in different tools for different purposes. In some of them we may be interested in expressing expected properties of the program if it were correct, i.e., intended properties, whereas in other contexts we may also be interested in expressing properties of the actual program in hand, i.e., actual properties, which may or may not correspond to the user's intention. For example, we can use program analysis techniques to infer properties of the program in hand and then use assertions in order to express the results of analysis. Thus, the assertion language should be able to express both intended and actual properties of programs. However, all the assertions presented in the examples in previous sections relate to intended properties. We have delayed the other uses of assertions until now for clarity of the presentation.

In our assertion language we allow adding in front of an assertion a flag which clearly identifies the *status* of the assertion. The status indicates whether the assertion refers to intended or actual properties, and possibly some additional information. Five different status are considered. We list them below, grouped according to who is usually the generator of such assertions:

– For assertions written by the user:
  `check` The assertion expresses an intended property. Note that the assertion may hold or not in the current version of the program.
  `trust` The assertion expresses an actual property. The difference with status `true` introduced below is that this information is given by the user and it may not be possible to infer it automatically.
– For assertions which are results of static analyses:
  `true` The assertion expresses an actual property of the current version of the program. Such property has been automatically inferred.
– For assertions which are the result of static checking:
  `checked` A `check` assertion which expresses an intended property is rewritten with the status `checked` during compile-time checking (see Chapter 2) when such property is proved to actually hold in the current version of the program for any valid initial query.
  `false` Similarly, a `check` assertion is rewritten with the status `false` during compile-time checking when such property is proved not to hold in the current version of the program for some valid initial query.

As already mentioned, all the assertions presented in the previous sections express intended properties and are assumed to be written by the user. Thus, they should have the status `check`. However, for pragmatic reasons, the status `check` is considered optional and if no status is given, `check` is assumed by default. For example, the assertion:

```
:- check success p(X) : ground(X).
```

can also be written "`:- success p(X) : ground(X).`"

Note also that the program-point assertions seen in Section 1.3.5 were introduced in the program as literals of the `check/1` predicate. This is because their status is `check`. If, however, we would like to add a program-point assertion with a different status we simply replace `check` by the corresponding status (`true`, `trust`, `checked` or `false`). See Section 1.10.1 for syntactic details. Again, if we want to execute a program with program-point assertions we can simply define the predicate corresponding to the status so that it always succeeds. For example, if the status is `true` we then define the predicate `true/1` (resp., `trust/1`, `checked/1`, `false/1`) as "`true(_).`."

Example 1.7.1. Figure 1.2 presents the same program as in Figure 1.1 but rather than with `check` assertions, with both predicate and program-point `true` assertions which express analysis results. The results have been generated by CiaoPP [1.6, 1.16] using goal-dependent mode analysis. Predicate and/or program-point assertions may be generated according to the user's choice. Program-point assertions contain information for each program point and are literals of the `true/1` predicate. Regarding predicate assertions, for conciseness, compound (`pred`) predicate assertions are usually produced by the analyser. Compound assertions will be introduced in Section 1.10.2.

```
:- entry qsort(L,R) : ground(L).
:- true pred qsort(A,B) : ground(A) => ( ground(A), ground(B) ).

qsort([X|L],R) :-
        true((ground([L,X]),var(L1),var(L2),var(R1),var(R2))),
        partition(L,X,L1,L2),
        true((ground([L,L1,L2,X]),var(R1),var(R2))),
        qsort(L2,R2),
        true((ground([L,L1,L2,R2,X]),var(R1))),
        qsort(L1,R1),
        true(ground([L,L1,L2,R1,R2,X])),
        append(R1,[X|R2],R),
        true(ground([L,L1,L2,R,R1,R2,X])).
qsort([],[]).

:- true pred partition(A,B,C,D)
         : ( ground(A), ground(B), var(C), var(D) )
        => ( ground(A), ground(B), ground(C), ground(D) ).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        true((ground([C,E,R]),var(Left1),var(Right))),
        E<C, !,
        true((ground([C,E,R]),var(Left1),var(Right))),
        partition(R,C,Left1,Right),
        true(ground([C,E,Left1,R,Right])).
partition([E|R],C,Left,[E|Right1]) :-
        true((ground([C,E,R]),var(Left),var(Right1))),
        E>=C,
        true((ground([C,E,R]),var(Left),var(Right1))),
        partition(R,C,Left,Right1),
        true(ground([C,E,Left,R,Right1])).
```

Fig. 1.2. Analysis results expressed as assertions

Though both `true` and `trust` assertions refer to properties of the actual program it is important to see that they are not equivalent. As already mentioned, `true` assertions are generated by analysis and are automatically provable, whereas `trust` assertions are provided by the user and often they are not provable either because part of the program is not available or because analysis is not powerful enough. In any case, analysis is instructed to *trust* such assertions.

The status `trust` receives this name for historical reasons. In [1.4] `trust` declarations were already used in order to provide the analyser with additional information so that it could improve its results. Note that a `trust` assertion for a predicate p may not only improve the analysis information for the predicate p (if the information it contains is better than that generated by analysis) but also it may allow improving the analysis information on other predicates which depend on p.

Example 1.7.2. Consider the following program in which the definition of predicate `r/2` is not available but where there is a `trust` assertion which states that upon success of `r(A,B)`, `B` is a list provided that `A` was a list on call:

```
:- entry p(X,Y).
:- trust success r(A,B) : list(A) => list(B).

p(X,Y):- q(X), r(X,Y).

q([])
q([_|Xs]):- q(Xs).
```

Assume we are using a goal-dependent type analyser. The `entry` assertion for predicate `p/2` informs the analyser that such predicate can be subject to initial queries. In addition, the analyser assumes that the set of existing `entry` assertions cover all possible initial queries. Since there is no `entry` for predicates `q` nor `r`, the analyser assumes that these predicates cannot be called in initial queries. Without the `trust` assertion, typically analysis would infer the following call and success types for predicates `p` and `r`:

```
:- true pred p(A,B) : (term(A),term(B)) => (list(A),term(B)).
:- true pred r(A,B) : (list(A),term(B)) => (list(A),term(B)).
```

where the type `term` represents the set of all possible terms. However, by exploiting the information in the `trust` assertion analysis can conclude that:

```
:- true pred p(A,B) : (term(A),term(B)) => (list(A),list(B)).
:- true pred r(A,B) : (list(A),term(B)) => (list(A),list(B)).
```

Note that not only the information on `r` has been improved, but also that of `p` since it depends on `r`.

It is important to mention that even though `trust` assertions are trusted by the analyser to improve its information unless they are incompatible with the information generated by the analyser (see [1.4]), they may also be subject to run-time checking. The translation scheme for assertions with the status `trust` is exactly the same as the one given in Chapter 2 for assertions with the status `check`. It should be an option whether only `check` assertions or both `check` and `trust` assertions should be checked at run-time.

A similar situation happens with `entry` assertions (presented in Section 1.3.4). If analysis is goal-dependent, it assumes that the entries hold, and thus all the information generated is correct under this assumption, but such information may become incorrect if the run-time queries do not conform to the existing `entry` assertions. Thus, in order to guarantee that the results of static checking and/or program optimisation based on analysis results are sound we may also check entries at run-time. As in the case of

`trust` assertions, it should be an option of the compiler whether to perform run-time checking of `entry` assertions or not.[6]

1.8 An Assertion Schema for Computations

Though the assertions already presented for operational semantics, declarative semantics, and for reasoning about completeness are very useful, there are many other interesting properties of programs which cannot be expressed using the presented assertion schemas. This is why we introduce yet another assertion schema named `comp`, which relates to *computations*, where by computation we mean the (ordered) execution tree of all derivations of a goal from a calling state.

The `comp` schema is, in the same way as `success` and `calls` schemas, associated to predicates and is inherently operational. The `success` and `calls` schemas allow expressing properties about the execution states both when the predicate is called and when it terminates its execution with success. However, as we mentioned above, many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not expressible with such schemas. In particular, no property which refers to (a sequence of) intermediate states in the computation of the predicate can be (easily) expressed using `calls` and `success` predicate assertions only. Examples of properties of the computation which we may be interested in are: non-failure, termination, determinacy, non-suspension, non-floundering, etc. In our language, this sort of properties are expressed using the schema:

`:- comp` *Pred* [`:` *Precond*] `+` *Comp-prop*`.`

where *Pred* is a predicate descriptor, *Precond* is a logic formula on execution states, and *Comp-prop* is a logic formula on computations. As in the case of `success` assertions, the field '`:` *Precond*' is optional. An assertion built using the `comp` schema should be interpreted as "in any activation of *Pred* if *Precond* holds in the calling state then *Comp-prop* should also hold for the computation of *Pred*." Alternatively, it can be interpreted as "the applicability set of the assertion is the set of computations of *Pred* in which the logic formula on states *Precond* holds at the calling state, and its satisfiability set has all computations in which the logic formula on computations *Comp-prop* holds."

Example 1.8.1. The following assertion could be used to express that all computations of predicate `qsort` with the first argument being a list of numbers and the second an unconstrained variable at the calling state should produce

[6] The introduction of run-time tests into the original program is analogous to that performed for `calls` assertions but is only applied to initial calls to the program (via a transformation of the program which renames apart the internal calls).

at least one solution in finite time (the property of the computation `succeeds` will be further discussed in Section 1.9.3):

```
:- comp qsort(L,R) : ( list(L,num), var(R) ) + succeeds.
```

where the atom `succeeds` is implicitly interpreted as `succeeds(qsort(L,R))`, with an extra argument, i.e., it is the execution of `qsort(L,R)` that has to succeed.

1.8.1 Logic Formulae about Computations

Similarly to logic formulae about execution states, logic formulae about computations can be conjunctions and/or disjunctions of formulae, where the atomic formulae are property predicates (about computations). As before, conjunctions and disjunctions are written in CLP syntax (i.e., are commas and semicolons, respectively).

As in the case of logic formulae about execution states, given a `comp` assertion for *Pred* with logic formula *Comp-prop* on computations and an execution state for a goal of the predicate of *Pred* in calling store θ, we first apply a renaming on the assertion which relates the variables in *Pred* with the variables in θ. Then we evaluate each atomic formula AFC in *Comp-prop* and the evaluation of *Comp-prop* is obtained by composing the values obtained for each AFC and simplifying the resulting expression. As before, we denote by $eval(AFC, \theta, P, IS)$ the evaluation of AFC in θ by IS w.r.t. the definition of properties P. Note that, in general, properties of the computation cannot be decided by looking at the store θ alone, as it is the case with properties of execution states. Thus, IS may need to reconstruct (part of) the computation of a particular instance of $Pred$, according to the calling store θ, in order to decide whether AFC holds or not. We assume that P contains, in addition to the definition of the property predicates, the program under consideration, so that reconstructing the computations is possible. Since the necessary part of the computation required may be an infinite object, it is possible that the process of reconstructing such computation does not terminate, in which case $eval(AFC, \theta, P, IS)$ will not terminate either. Thus, we admit that the evaluation of an atom of a property of the computation AFC, in addition to returning *true*, *false* or AFC (if IS cannot decide the property), may also not terminate, precisely in those cases in which the execution of $\langle Pred \mid \theta \rangle$ does not terminate either. We argue that this is admissible since the evaluation of the property still does not introduce non-termination, in the sense that if the program execution terminates the evaluation of properties will also terminate.

Example 1.8.2. Consider again the `comp` assertion in Example 1.8.1. Consider also that during dynamic checking of such assertion we reach an execution state of the form $\langle$`qsort(X,Y)` :: *Goal* $\mid \theta\rangle$, i.e., `qsort(X,Y)` is the

first literal to solve and *Goal* is a (possibly empty) conjunction of literals, where θ indicates that `X` takes the value `[1,5,3]` and `Y` is an unconstrained free variable. In such state our `comp` assertion is applicable since $eval($`list([1,5,3],int)`$, \theta, P, IS)$ and $eval($`var(Y)`$), \theta, P, IS)$, where P must contain at least the definition of the parametric property `list/2` (and of `var/1` if it were not a built-in predicate), can be proved to take the value *true* using an appropriate IS. In order to see whether the assertion is satisfiable we have to compute the value of $eval($`succeeds(qsort([1,5,3],Y))`$, \theta, P, IS)$, where P must contain at least the definition of `succeeds/1` and also of `qsort/2` and of all other predicates it uses, in this case `partition/4` and `append/3`. In our example the computation which has to be reconstructed is $\langle$`qsort([1,5,3],Y)` $\mid \theta\rangle$, which is finite. Thus, checking the property should terminate.

1.9 Defining Property Predicates

All our assertion schemas are parameterised on logic formulae for expressing the particular properties of the execution states (in `calls` and `success` assertions), of the correct answers (in `inmodel` assertions), and of the computations (in `comp` assertions). Atoms in such formulae are of the predicates which we call *property predicates* (or *properties* for short when the context is clear enough). In this section we discuss how to define such property predicates. We have not presented the property predicates allowed in our assertion language yet because the assertion language is parametric w.r.t. the set of property predicates of interest. Thus, rather than having a fixed set of such predicates, we allow users to define their own properties in a very flexible way.

Since we have assumed that our source language is a logic and/or constraint logic programming language, in which it is natural to define predicates, it also seems natural to use the underlying CLP language to define the property predicates. This design decision has very important implications: (1) The user does not need to learn a new language for defining property predicates since the same language used for writing programs can be used. (2) This makes the assertion language extremely expressive since the user can define almost any predicate property that is considered of interest when dealing with a particular program. (3) With a little run-time support, atomic logic formulae in assertions can be evaluated by simply executing them on the underlying CLP system. This can be seen as taking the underlying CLP system as an inference system.[7] (4) Though the assertion language may remain decidable for run-time checking under some sensible restrictions on the property predicates used (such as that their execution always terminates), there

[7] The use of *executable* properties which can be checked dynamically, i.e., by executing the code defining them, has also been proposed in the context of declarative debugging [1.11].

is little hope that the assertion language remains decidable for compile-time checking given any fixed set of static inference systems available, as we allow users to define their own predicates. This is why systems designed with particular inference systems in mind generally only allow using a predefined set of property predicates or they have a very restricted language for defining new property predicates, which is a restriction that we want to lift.

1.9.1 Declaring Property Predicates

Since the set of property predicates is not fixed in our assertion language, in order to be able to perform some syntactic checking on the assertions given for a program, i.e., to check whether they are consistently written, we require that all predicates which can be used as property predicates are declared as such using an assertion of the form:

`:-` [*IS_List*] `prop` *Pred-Spec*. or `:-` [*IS_List*] `cprop` *Pred-Spec*.

where *Pred-Spec* is a term of the form *p*/*n*, where *p* is a predicate symbol and *n* its arity. They indicate, respectively, that the predicate can be used in logic formula about states or about computations. Also, the (optional) field *IS_List* contains a list of the inference systems in which *Pred* is provable. We consider that it is not required to indicate whether the underlying CLP system can be used to prove the property or not: if a predicate property is defined in the CLP source language, such definition is assumed to be exact and thus the underlying CLP system can be used to decide whether the property holds or not. As further discussed in Chapter 2, we impose some restrictions on the code which defines property predicates. Thus, we also assume that the definition in source code of any property predicate for which a `prop` or `cprop` declaration exists satisfies such restrictions.

All property predicates which may appear in the logic formulae of the assertions given for the program must be declared, independently of whether a definition as source code is given for them or not. In particular, we should declare as property predicates those ones for which there is an inference system which partially captures (or is complete for) them and which we intend to use in assertions.

Example 1.9.1. Consider the property predicate about states `ground/1`. An argument satisfies this property if it is bound to a Herbrand term without variables. This property is often provable using static inference systems which reason about variable groundness. Consider now that in our tool there are two different inference systems called *sharing* and *def* which are capable of proving the property `ground/1`. This should be indicated with a declaration like '`:- [sharing, def] prop ground/1.`'. In addition, it is likely that a built-in (or library) predicate with the same name and meaning exists in the CLP system. In that case, the property can be decided during dynamic checking using the underlying CLP system. If it is not a built-in we could write a definition of such predicate in CLP, as further discussed below.

1.9.2 Defining Property Predicates for Execution States

In this section we discuss by means of examples several issues related to the definition of property predicates about execution states by means of CLP programs. We start with an example.

Example 1.9.2. Consider the following definition of the property predicate `list/1` by means of a regular program [1.32, 1.10]:

```
list([]).
list([_|Xs]):- list(Xs).
```

The above definition can be used to dynamically decide whether a term is of type `list` or not. This case is also interesting because if an inference system which captures regular types (which are further discussed in this book in Chapter 4) is available, then it may be able to prove such property statically by using the code above, which the inference system can safely handle as a type declaration. This should be indicated with the assertion '`:- regtype prop list/1.`' assuming that `regtype` is the name of the inference system for regular types. This is an example of how users can define new regular types in our assertion language by means of a (regular) CLP program.

A distinguishing feature of (constraint) logic programming w.r.t other programming paradigms such as functional or imperative programming is that, in a given execution state, the values of certain program variables may be (partially) undefined. This is a consequence of the existence of "logical variables" whose value may be further instantiated during forward execution. This feature has to be taken into account when defining properties of execution states.

Example 1.9.3. In the definition of property `list` in Example 1.9.2 above it is not obvious which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list" (let us indicate this property with the property predicate `inst_to_list`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list" (we will associate this property with the property predicate `compat_with_list`). For example, `inst_to_list` should be true for the terms `[]`, `[1,2]`, and `[X,Y]`, but should not for `X` and `[a|X]`. In turn, `compat_with_list` should be true for `[]`, `X`, `[1,2]`, and `[a|X]`, but should not be for `[a|1]` and `a`.

We refer to properties such as `inst_to_list` above as *instantiation properties* and to those such as `compat_with_list` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types"), and to the corresponding property predicates as instantiation and compatibility property predicates.

It turns out that both of these notions are quite useful in practice. Consider for example a definition of the well known predicate `append`:

```
append([],L,L).
append([X|Xs],L,[X|NL]):- append(Xs,L,NL).
```

For this predicate we probably would like to use `compat_with_list` to state that in all calls to `append` all three arguments must be compatible with lists in an assertion like:

```
:- calls append(A,B,C) :
   (compat_with_list(A), compat_with_list(B), compat_with_list(C)).
```

With this assertion, no error will be flagged for a call to `append` such as `append([2],L,R)`, since `L` can be instantiated to a list later on the execution, but a call `append([],a,R)` would indeed flag an error.

On the other hand, we probably would like to also use `inst_to_list` to describe the type of calls for which `qsort` has been designed, i.e., those in which the first argument must indeed be a list. This was done for example in assertion `A1` of Figure 1.1:

```
:- calls qsort(L,R) : list(L).
```

i.e., here we clearly wanted `list(L)` to mean `inst_to_list(L)`.

Since both kinds of properties are properties of interest, one possibility is to define, in each case, two distinct property predicates, one of each kind. This will force us to define both `inst_to_list` and `compat_with_list` in different ways.

Example 1.9.4. A possible definition of `inst_to_list` is the following:

```
:- prop inst_to_list/1.

inst_to_list(X) :-
       nonvar(X), inst_to_list_aux(X).

inst_to_list_aux([]).
inst_to_list_aux([_|T]) :- inst_to_list(T).
```

However, one would like that the more natural definition of `list` of Example 1.9.2 could be used both as an instantiation and as a compatibility property, by simply instructing the system to handle the definition of the property in the appropriate way. We introduce a mechanism in our assertion language for this purpose. Properties about execution states are interpreted by default as instantiation properties. Thus, writing:

```
:- calls qsort(L,R) : list(L).
```

has the desired effect. If we are interested in using the definition of a predicate *Property* as a compatibility property, we should indicate it in the formula as `compat(`*Property*`)` and it will automatically be interpreted as: "*Property* holds in the current store or it can be made to hold by adding bindings (or constraints) to the current store." Thus, writing:

```
:- calls append(A,B,C)
        : ( compat(list(A)), compat(list(B)), compat(list(C)) ).
```

also has the desired effect.

This allows to define property predicates as compatibility properties, and use them as either an instantiation or a compatibility property on states. However, note that if the definition of a state property predicate *Prop* contains certain impure built-in predicates which explicitly (e.g., `nonvar`, `var`) or implicitly (e.g., `integer`, `atom`, `>`) perform some degree of instantiation checking, it may not be correct to use *Prop* as a compatibility property.

On the other hand, note that the definition of `list` would not behave as expected either if used as an instantiation property as is. This is because if we execute it in a store which is not sufficiently instantiated, execution will succeed and add the necessary bindings or constraints for the property to hold rather than failing, which is what the instantiation property should do. On the contrary, `list` behaves as expected of a compatibility property. Nonetheless, we have preferred to consider properties such as `list` instantiation properties by default. This is because we prefer to put the burden of interpreting the definition of properties as instantiation properties on the inference system, rather than putting such burden on the users by forcing them to write the instantiation property explicitly. We argue that this is a natural choice, since in most cases writing down the definition of the compatibility property is easier, but (at least in our experience) in a large number of logic formulae about execution states users are interested in the instantiation property rather than in the compatibility property.

1.9.3 Defining Property Predicates for Computations

Since existing (constraint) logic programming systems have meta programming facilities which allow having atoms (calls to predicates) appearing as arguments of calls to other predicates, it is in principle possible to use the underlying programming language in order to also define property predicates for computations. However, this is not as easy as defining property predicates for execution states.

Example 1.9.5. Consider the property of the computation `succeeds(`*Pred*`)` which is to be interpreted as "the computation of *Pred* in the current store produces at least one solution in finite time." This property, which when appears in a `comp` assertion has no arguments, has to be defined in CLP as a predicate with one argument (which receives the goal on whose computation we aim at checking the property). The following definition could be used:

```
succeeds(Goal):- call(Goal).
```

Although defining the above predicate property in CLP has not been very difficult, in general, defining a property of a computation in source code

requires to actually perform the computation (which in the example above is done using the `call` built-in predicate). This is possible since checking the property does not need to observe intermediate states of the computation of `qsort(A,B)` (with the values of `A` and `B` according to the store) and it suffices to check whether the call succeeds or not. For other properties which may need observing internal states of the execution we may need to program a meta-interpreter. This has the disadvantage that the property being defined may end up being rather obscure.

An additional difficulty in expressing property predicates for computations as CLP predicates is that many of the interesting properties of computations we may want to write are by nature undecidable. Thus, it is just not possible to provide an effective definition of such properties which can be executed during dynamic checking. A good example of this is trying to define a predicate `terminates/1` which decides whether computation of a predicate (universally) terminates or not. Clearly, it is not possible to provide a general definition of such predicate. Thus, `comp` assertions are often not amenable to dynamic assertion checking, since it is difficult to write the properties involved in such a way that they can be executed efficiently and non trivial results obtained. However, we still include assertions about properties of computations in our language for several reasons: (1) when considering static assertion checking, it is often the case that there are analysis which obtain useful results on properties which are undecidable in general. This is because such analyses use sufficient conditions which guarantee that the properties hold of the program. For example, a good number of termination analyses exist which can prove termination of a high percentage of terminating programs, though there will always be terminating programs which cannot be proved to terminate. (2) In other cases, approximate definitions in source code can often be provided for dynamic and/or static checking of properties of the computation. This is discussed in the following section. (3) They are also useful for expressing the results of static analyses which infer properties of computations such as termination, non-failure, determinacy,... (4) They can be used for expressing properties which we do not aim at checking but rather they are interesting for documentation purposes. An example of this is the Ciao reference manual [1.3] where, for example, the assertion '`:- trust comp write/1 + iso.`' indicates that the implementation of predicate `write/1` behaves according to the ISO Prolog specification.

1.9.4 Approximating Property Predicates

As already mentioned, in our assertion language we do not require an exact definition of every predicate property used. This is useful for at least two reasons. One reason is that there are a good number of interesting properties which are not decidable and for which it is just not possible to provide exact definitions using the source CLP language. As discussed above, this is often the case when trying to define property predicates about computations.

Another reason is that sometimes the user prefers not to provide an exact definition because even though it is possible, it is a hard or tedious task. However, it is very simple to provide an approximate but still useful definition. The user may decide to provide an exact definition at a later point in time if so desired.

Our assertion language, rather than restricting the set of property predicates to those for which an exact definition in the source language is provided or simply returning AF as the result of $eval(AF, \theta, P, IS)$ if IS does not capture the property AF, it allows providing *approximations* of such property predicates AF. Such approximations provide sufficient conditions for returning *true* (proving AF) or *false* (disproving AF). These approximations are given using assertions of the form:

`:- proves` *Pred* `:` *Cond*. or `:- disproves` *Pred* `:` *Cond*.

where *Pred* is a property predicate descriptor and *Cond* a logic formula about states or about computations. They indicate that given a current store θ if we can prove that *Cond* holds in θ (using any inference system) then we have also proved that *Pred* holds in θ, if a `proves` assertion is given, or that *Pred* does not hold in θ, if a `disproves` assertion is given.

Example 1.9.6. Consider a static inference system called `simple_stat` which is capable of determining that at certain program points some arguments are bound to integer numbers. We can use the following declarations:

```
:- simple_stat prop integer/1.

:- proves ground(X) : integer(X).
:- disproves var(X) : integer(X).
```

in order to indicate that (1) `simple_stat` can prove the state property `integer/1` and (2) the fact that we can establish that some argument is an integer number can be used to guarantee that such argument is ground and also to prove that such argument is definitely not a free variable. Note that the `proves` and `disproves` assertions are independent of any inference system. This means that they would also be useful if some other inference system were able to prove the property `integer/1`. Thus, though the predicate properties `ground` and `var` are not directly inferred by `simple_stat` we can use them in assertions and still be able to either prove or disprove such assertions, in this case statically, using `simple_stat`.

1.10 Syntax of and Extensions to the Assertion Language

In this section we provide a summary of the syntax of assertions. We then introduce another predicate assertion schema which can be used in addition

to the ones introduced previously. It can be seen as syntactic sugar for a set of predicate assertions. Finally we comment on some other syntactic sugar which facilitates the writing of assertions.

1.10.1 Syntax of the Assertion Language

We now summarise the syntax of the assertions presented with the following two formal grammars. The first one defines the syntax of program assertions, from the non-terminal *program-assert*:

program-assert	::=	*predicate-assert*
	|	*prog-point-assert*
predicate-assert	::=	`:-` *stat-flag pred-assert* `.`
	|	`:-` *entry* `.`
pred-assert	::=	`calls` *pred-cond*
	|	`success` *pred-cond direction state-log-formula*
	|	`inmodel` *pred-desc direction state-log-formula*
	|	`comp` *pred-cond* `+` *comp-log-formula*
entry	::=	`entry` *pred-cond*
pred-cond	::=	*pred-desc*
	|	*pred-desc* : *state-log-formula*
pred-desc	::=	*Pred-name*
	|	*Pred-name*`(`*args*`)`
args	::=	*Var*
	|	*Var*`,` *args*
state-log-formula	::=	`(`*state-log-formula* `,` *state-log-formula*`)`
	|	`(`*state-log-formula* `;` *state-log-formula*`)`
	|	`compat(`*State-prop*`)`
	|	*State-prop*
comp-log-formula	::=	*comp-log-formula* `,` *comp-log-formula*
	|	*comp-log-formula* `;` *comp-log-formula*
	|	*Comp-prop*
stat-flag	::=	*status*
	|	ϵ
status	::=	`check`
	|	`true`
	|	`checked`
	|	`trust`
	|	`false`
direction	::=	`=>`
	|	`<=`
prog-point-assert	::=	*status*`(`*state-log-formula*`)`

There are some non-terminals in the grammar which are not defined. This is because they are constraint-domain and/or platform dependent. They can

be easily distinguished in the previous grammar because their name starts with a capital letter:

Pred-name As we are interested in having an assertion language which looks homogeneous with the CLP language used, we admit as *Pred-name* any valid name for a predicate in the underlying CLP language. Usually, non-empty strings of characters which start with a lower-case letter.

Var It corresponds to the syntax for variables in the CLP language. Usually, non-empty strings of characters which start with a capital letter. As mentioned before, it is assumed that all variables in the same predicate description are distinct.

State-prop An atom of a `prop` property predicate.

Comp-prop An atom of a `cprop` property predicate.

The following grammar defines the syntax of assertions for declaring property predicates, from the non-terminal *prop-assert*:

prop-assert	::=	*prop-exp*
		| *approx-exp*
prop-exp	::=	`:-` *is-flag prop pred-spec* `.`
prop	::=	`prop`
		| `cprop`
is-flag	::=	`[` *is-idlist* `]`
		| *Is-id*
		| ϵ
is-idlist	::=	*Is-id* `,` *is-idlist*
		| *Is-id*
pred-spec	::=	*Pred-name*/*Number*
approx-assert	::=	`:-` *approx approx-exp* `.`
approx	::=	`proves`
		| `disproves`
approx-exp	::=	*State-prop* `:` *state-log-formula*
		| *Comp-prop* `:` *comp-log-formula*

The new non-terminals in the grammar are as follows:

Is-id A constant of the language which uniquely identifies an inference system in the debugging system being used.

Number A number (which is meant to be the arity of a predicate).

1.10.2 Grouping Assertions: Compound Assertions

The motivation for introducing compound assertions is twofold. First, when more than one `success` (resp. `comp`) assertion is given by the user for the same predicate, in the user's mind this set is usually meant to cover all the

Table 1.1. Transforming compound into basic assertions.

Field	Translation if given	Otherwise
`=>` *Postcond*	`success` *Pred* `:` *Precond* `=>` *Postcond*	$\emptyset$
`+` *Comp-prop*	`comp` *Pred* `:` *Precond* `+` *Comp-prop*	$\emptyset$

different uses of the predicate. In such cases, the disjunction of the preconditions in all the `success` (resp. `comp` assertions) is often a description of the possible calls to the predicate. However, the user would have to explicitly write down a `calls` assertion to express this. It would be desirable to have the `calls` assertion be automatically generated in such cases for the set of assertions, rather than having to add it manually. Compound assertions allow this. Second, a disadvantage of the assertion schemas presented in sections 1.3 and 1.8 is that it is often the case that in order to express a series of properties of a predicate, several of them need to be written.

Each compound assertion is translated into one, two, or even three basic predicate assertions, depending on how many of the fields in the compound assertion are given. Compound assertions are built using the `pred` schema, which has the form:[8]

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*] .

Example 1.10.1. The following assertion indicates that whenever we call `qsort` with the first argument being a list, the computation should terminate and if the computation succeeds, on termination the second argument should also be a list.

```
:- pred qsort(L,R) : list(L) => list(R) + terminates.
```

in addition, if this is the only `pred` assertion given for predicate `qsort`, then it also indicates that all calls to `qsort` should have a list in the first argument.

Table 1.1 presents how a compound assertion is translated into basic `success` and `comp` assertions. Generation of `calls` assertions from compound assertions is more involved, as the set of all compound assertions for one predicate must cover all possible calls to that predicate. Thus, if the set of compound assertions for a predicate *Pred* is $\{A_1, \ldots, A_n\}$, let A_i =*Pred* `:` C_i [`=>` S_i] [`+` $Comp_i$], then the (only) `calls` assertion which is generated is

`:- calls` *Pred*: $\vee_{i=1}^{n} C_i$.

Example 1.10.2. Consider the two following compound assertions for predicate `qsort`:

[8] Note that the syntax grammar presented previously does not incclude this extension.

```
:- pred qsort(A,B) : numlist(A) => numlist(B) + terminates.
:- pred qsort(A,B) : intlist(A) => intlist(B) + terminates.
```

The `calls` basic assertion which would be generated is:

```
:- calls pred qsort(A,B) : (numlist(A) ; intlist(A)).
```

Note that when compound assertions are used, a `calls` assertion is always implicitly generated. If we do not want the `calls` assertion to be generated (for example because the set of assertions available does not cover all possible uses of the predicate) basic `success` or `comp` assertions rather than compound (`pred`) assertions should be used.

1.10.3 Some Additional Syntactic Sugar

There are a number of syntactic sugar conventions in the assertion language which can be added to facilitate the writing of assertions. We mention here, by means of examples, some of the most interesting ones.[9]

Abridged syntax. With this syntax it is not necessary to explicitly mention which argument of the predicate descriptor the logic formulae refer to. The different arguments are identified by position. Individual logic formulae are separated by `*` and they refer to the predicate arguments by order. For example:

```
:- calls qsort(A,B) : list * term.
:- calls qsort/2 : list * term.
```

are both equivalent to:

```
:- calls qsort(A,B) : ( list(A), term(B) ).
```

When there is the need for associating two or more properties to the same argument then the following syntax may be used:

```
:- calls qsort/2 : { list, ground } * term.
```

which is equivalent to:

```
:- calls qsort(A,B) : ( list(A), ground(A), term(B) ).
```

The abridged syntax can be mixed with the normal syntax in a given assertion, provided that each "field" of the assertion is written using only one syntax. For example:

```
:- success qsort(A,B) : list * term => list(B).
```

[9] Note that the syntax grammar presented previously does not incorporate these extensions.

Compatible properties. In some cases, the programmer wants to specify compatibility properties of some arguments both at the call and success states of the predicate. To avoid repeating the properties, the syntax of the following example can be used:

```
:- pred qsort(A,B) :: ( list(A), list(B) ).
```

which is equivalent to:

```
:- pred qsort(A,B) : ( compat(list(A)), compat(list(B)) ) =>
                     ( compat(list(A)), compat(list(B)) ).
```

This kind of writing can also be "in-lined" into the predicate arguments. For example, the following assertion is equivalent to the two ones above:

```
:- pred qsort(list,list).
```

Modes. They allow specifying in a compact way several properties which refer to one argument. Thus, modes can be seen as property macros. For example, provided that the following mode definition exists:

```
:- modedef out(X) : var(X) => ground(X).
```

then, instead of the first assertion below, the second one could be written, which is equivalent:

```
:- pred qsort(A,B) : var(B) => ground(B).
:- pred qsort(A,out(B)).
```

Our assertion language generalises the classical concept of modes, allowing users to define their own. For example, the classical Prolog modes "+" (i.e., the corresponding argument is a non-variable on input) and "−" (the argument is a variable on input) can be expressed in our language by defining them as:

```
:- modedef '+'(X) : nonvar(X).
:- modedef '-'(X) : var(X).
```

Mode syntax can be mixed with any other syntax, as in the first assertion below (which is equivalent to the second one):

```
:- pred qsort(list,out(B)) => list(B).
:- pred qsort(A,B) :: list(A) : var(B) => ( list(B), ground(B) ).
```

Also, "meta-"modes can be defined which allow writing assertions in a very compact way. The previous assertion could be written as follows, by using a different mode definition, i.e.:

```
:- modedef out(X,P) : var(X) => ( P(X), ground(X) ).
:- pred qsort(list,out(list)).
```

where `P(X)` is a higher-order notation used in Ciao which stands for applying the property predicate `P`, whatever value it has (in this example `list`) to the argument `X`.

1.11 Discussion

In this chapter we have presented an assertion language which should be of interest in several different tools and for different purposes: compile-time checking, replacing the oracle in declarative debugging, run-time checking, providing information to the optimiser, general communication with the compiler, and automatic documentation. In such a situation it is difficult to restrict beforehand the properties of a program which we may be interested in expressing by means of assertions. Also, we cannot assume the existence of a particular inference system in every tool other than the underlying CLP system.

An assertion language can be seen as composed of: (1) a set of assertion schemas, (2) a syntax to build logic formulae for such schemas, and (3) a syntax to define predicates for the atomic logic formulae. In addition to the assertion language we need some inference system, capable of evaluating the assertions for a program. Often, assertion languages are designed bottom-up, in the sense that once the inference system to be used has been decided, the three components mentioned are defined so that the assertion language remains *decidable*, in the sense that any assertion expressible in the assertion language can be either proved to hold or not to hold in a program using the available inference system. This allows debugging systems based on this approach to reject programs which have not been validated w.r.t. the given assertions. In contrast, the design of our assertion language is top-down in the sense that it is not induced by any particular inference system. We do not assume a fixed set of property predicates but rather we provide the means for defining new property predicates. This can be done by providing exact descriptions of properties as CLP predicates and also by using assertions which indicate that a (possibly built-in) predicate property can be proved by a given inference system. We also use assertions to provide sufficient conditions for proving and disproving predicate properties when such exact definition is not available. As a result of this, on one hand our assertion language is very flexible, and on the other hand we have to lift the assumption that any assertion is decidable in the system. In other words, we have to live with the possible undecidability of any logic formula and thus of any assertion. Thus, in the tools which use the proposed assertion language we have to be able to deal safely with approximations [1.5]. We argue that lifting the decidability assumption opens the door to very interesting possibilities and that still very useful results can be obtained by combining static and dynamic checking of the assertions.

Even though the properties given in assertions may not be decidable, it is our view that assertions should be checked as much as possible at compile-time via static analysis. The system should be able to make conservative approximations in the cases in which precise information cannot be inferred (and some assertions may remain unproven). This is the approach taken in Chapter 2. Note, however, that if the properties allowed in assertions are not

decidable the approach to the treatment of "don't know" during compile-time checking has to be weaker than the one used for "strong" debugging systems: in our case the program cannot be rejected. The case that the analysis is not capable either to prove nor disprove an assertion may be because we do not have an accurate enough inference system or simply because the assertion is just not statically decidable. In this we follow the spirit of [1.4, 1.8]. However, we do not rule out the definition and use of a decidable debugging system, e.g., based on types, if so desired.

Acknowledgements

The proposed assertion language is based on previous work by several DiSCiPl project members and numerous discussions within the project. However, any errors or omissions are only the fault of its authors. The authors would like to thank in particular Jan Małuszyński, Wlodek Drabent, and Pierre Deransart for many interesting discussions on assertions. Also Abder Aggoun, Helmut Simonis, Eric Vetillard and Claude Laï for their feedback on the kind of assertions needed in different state-of-the-art debugging tools.

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

1.1 A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20, 1994.

1.2 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

1.3 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.

1.4 F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

1.5 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyǹski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

1.6 F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, November 1999.

1.7 D. Cordes and M. Brown. The Literate Programming Paradigm. *IEEE Computer Magazine*, June 1991.

1.8 P. Cousot. Types as Abstract Interpretations. In *Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, January 1997.
1.9 P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
1.10 P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
1.11 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.
1.12 W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
1.13 Lisa Friendly. The Design of Distributed Hyperlink Program Documentation. In *Int'l. WS on Hypermedia Design*, Workshops in Computing. Springer, June 1996. Available from `http://java.sun.com/docs/javadoc-paper.html`.
1.14 M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.
1.15 M. Hermenegildo. A Documentation Generator for Logic Programming Systems. In *ICLP'99 Workshop on Logic Programming Environments*, pages 80–97. N.M. State University, December 1999.
1.16 M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
1.17 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
1.18 P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.
1.19 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
1.20 A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.
1.21 D. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984.
1.22 K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.
1.23 K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
1.24 L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

1.25 G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.
1.26 G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS. Springer-Verlag, 2000. To appear.
1.27 E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
1.28 Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.
1.29 P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.
1.30 E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.
1.31 R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
1.32 E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

2. A Generic Preprocessor for Program Validation and Debugging

Germán Puebla, Francisco Bueno, and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, S-28660-Madrid, Spain
email: {german,bueno,herme}@fi.upm.es

We present a generic preprocessor for combined static/dynamic validation and debugging of constraint logic programs. Passing programs through the preprocessor prior to execution allows detecting many bugs automatically. This is achieved by performing a repertoire of tests which range from simple syntactic checks to much more advanced checks based on static analysis of the program. Together with the program, the user may provide a series of assertions which trigger further automatic checking of the program. Such assertions are written using the assertion language presented in Chapter 1, which allows expressing a wide variety of properties. These properties extend beyond the predefined set which may be understandable by the available static analysers and include properties defined by means of user programs. In addition to user-provided assertions, in each particular CLP system assertions may be available for predefined system predicates. Checking of both user-provided assertions and assertions for system predicates is attempted first at compile-time by comparing them with the results of static analysis. This may allow statically proving that the assertions hold (i.e., they are validated) or that they are violated (and thus bugs detected). User-provided assertions (or parts of assertions) which cannot be statically proved nor disproved are optionally translated into run-time tests. The implementation of the preprocessor is generic in that it can be easily customised to different CLP systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. We also report on two tools which are instances of the generic preprocessor: CiaoPP (for the Ciao Prolog system) and CHIPRE (for the CHIP CLP(FD) system). The currently existing analyses include types, modes, non-failure, determinacy, and computational cost, and can treat modules separately, performing incremental analysis.

2.1 Introduction

Constraint Logic Programming (CLP) [2.34] is a powerful programming paradigm which allows solving hard combinatorial problems. As (constraint) logic programming systems mature and further and larger applications are built, an increased need arises for advanced development and debugging environments. In the current state of the practice, the tasks of validation and debugging of CLP programs are very costly in the software development pro-

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 63–107, 2000.

cess. This is especially true when the problems to be solved involve a large number of variables, constraints, and states.

As discussed in previous chapters, such advanced environments will likely comprise a variety of co-existing tools ranging from declarative debuggers to execution visualisers, such as those presented in this book. See also [2.24, 2.23] for a discussion on possible debugging scenarios. In order to have a satisfactory program we need at least the following two properties:

- The program is *correct*. During the development phase we often have programs which produce *wrong results* and/or fail to produce results for some valid input data, i.e., *missing results*. Also, the program may generate *run-time errors*. Though it is true that these problems mainly occur in the first stages of development, they may also appear later, for example, if the program is modified (perhaps to improve performance). The approach we propose is to use the generic preprocessor presented in this chapter for such problems.
- The program is reasonably *efficient*. Though a program may be correct, it may also be the case that the program takes too long to terminate for practical purposes. If this occurs, the program may need to be modified. For example, the constraint order, modelling of the problem, the solver being used, the heuristics applied, etc. may need to be changed. Though the preprocessor may be useful for efficiency debugging, for example using cost analysis, other tools are also of use, for example the visualisation tools presented in chapters 6 through 10.

Correctness checking, or checking for short, can be performed either at compile-time, i.e., before executing the program, or at run-time, i.e., while executing the program, or both. In both cases it is important that such checking be performed *automatically*. Also, the sooner in the development process an incorrect program is detected, the better. Thus, compile-time checking is generally preferable to run-time checking. In this chapter, we present a preprocessor for correctness checking which is automatic, generic (in the sense that it can be instantiated to different CLP languages), and it is based on a certain preference for compile-time checking over run-time checking. It is based on work previously presented in [2.39, 2.32, 2.31, 2.30].

2.1.1 Design of the Preprocessor

Most existing CLP systems perform correctness checks in one way or another. The classical scenario of such checking is depicted in Figure 2.1. Compile-time checking typically involves at least performing *syntactic checking*. Programs to be executed have to adhere to a given syntax. Those programs which do not satisfy such syntax are flagged at compile-time as *syntactically incorrect* instead of being executed, as depicted in Figure 2.1. Clearly, the fact that a program is syntactically correct does not guarantee that the program is

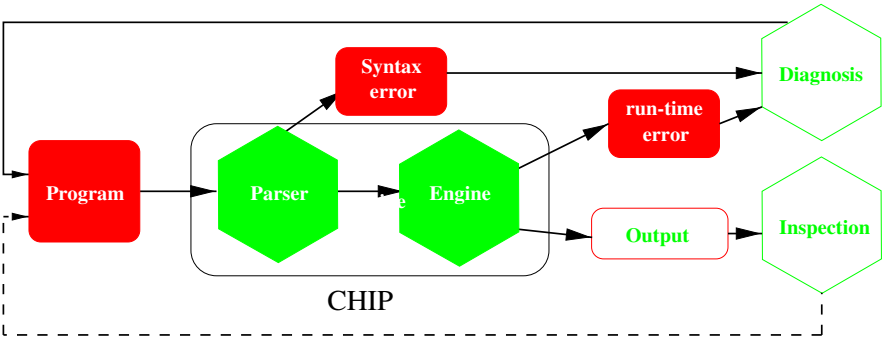


Fig. 2.1. Correctness Checking in CLP Systems.

correct. Run-time checking typically involves at least checking that built-in and library predicates are called as expected. If, for example, the types of the arguments are not those expected, the code for the predicate may not be valid and the possible results are not guaranteed to be correct. When some call to a built-in or library predicate is detected to be invalid, an error message is issued and generally execution aborted.

Example 2.1.1. Consider the following query and error message using CHIP:

```
22?- X=g,indomain(X).
Error 119 : domain variable expected in indomain(g)
```

where the CHIP built-in `indomain` assigns to its argument (`X` in this case) one of the values which remain available in its domain and enumerates all possible values on backtracking. However, this only makes sense if `X` is a finite domain variable. Clearly, this is not the case in the example, as `X` is bound to the constant `g`.

Unfortunately, the kinds of incorrectness errors which most existing CLP systems automatically detect either at compile-time or run-time are very limited. One reason for this is that the system does not have clear criteria to flag a program as incorrect other than syntax errors and invalid run-time calls to system predicates. This is because the system does not know the expected program behaviour which the user has in mind. A usual way to increase the criteria to detect incorrect programs is to provide the system with *assertions* (see Chapter 1) describing user's intentions, and which the system can check either at compile-time or at run-time.

The preprocessor we present greatly extends the checking capabilities of CLP systems in order to automatically detect more incorrectness symptoms. We have extended the traditional syntactic checks and also incorporated assertion-based checking both at compile-time and run-time. In our preprocessor, compile-time checking is based on a range of powerful program

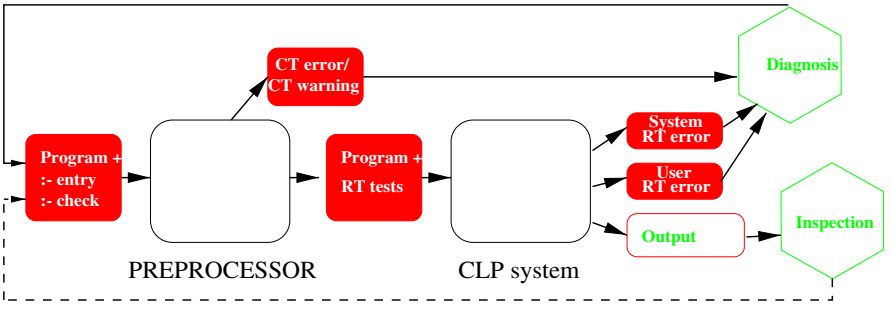


Fig. 2.2. Using the Preprocessor.

analysers, which gives the system a new flavour: sometimes (semantic) errors can be detected even without users having provided any assertion at all.

We call our system a *preprocessor* because programs are expected to be passed through it prior to being run on the CLP system. When a program is passed through the preprocessor, i.e., *preprocessed*, two kinds of messages may be produced: *error* messages and *warning* messages (depicted in Figure 2.2 as `CT error` and `CT warning`, respectively). Error messages are issued when evidence has been found that the program is definitely incorrect (i.e., an incorrectness symptom has been found). Whenever an error message is issued the program should be corrected, rather than executed. Warning messages are issued when the program is suspect of being incorrect. Thus, warning messages do not always mean that the program is incorrect, but they often help detecting bugs. Warning messages should also be taken into account in order to decide whether they correspond to bugs or not. If they do not, they can often be avoided by making slight modifications to the program. Thus, the next time the program is preprocessed such warning messages will not be issued any more. This is not strictly required, but avoids distracting messages which do not correspond to bugs. If messages are issued and the program modified, it should be preprocessed once again. Several iterations may be required until no more errors (nor warnings) are flagged.

As depicted in Figure 2.2, the starting point for assertion-based correctness debugging is a set of `check` assertions (see Chapter 1) which provide a (partial) description of the intended behaviour of the program. Besides `check` assertions, the programmer may also supply `entry` assertions for the program, which describe the valid queries to the program.

A fundamental design principle in our preprocessor is to be as unrestrictive as possible with regards to what the user needs to provide. To start with, and as mentioned in Chapter 1, the assertion language allows expressing properties which are much more general than, for example, traditional type declarations, and such that it may be undecidable whether they hold or not for a given program. Once we lift the requirement that assertions be

decidable, it is also natural to allow assertions to be *optional*: specifications may be given only for some parts of the program and even for those parts the information given may be incomplete. This allows preprocessing existing programs without having to add assertions to them and still being able to detect errors. And if we decide to add some assertions to our programs, such assertions may be given for only some procedures or program points, and for a given predicate we may perhaps have the type of one argument, the mode of another, and no information on other arguments. Finally, we would like the system to be able to *generate* assertions which describe the behaviour of the existing program, especially for parts of the program for which there are no `check` assertions. These assertions will have the status `true` (see Chapter 1) and can be visually inspected by the user for checking correctness.[1]

Note that, although neither `entry` nor `check` assertions are strictly required, the more effort the user invests in providing accurate ones, the more bugs can be automatically detected. Also, since `check` assertions may not encode a complete specification of the program, the fact that all `check` assertions hold does not necessarily mean that the program is correct, i.e., that it behaves according to the user's intention. It may be the case that the program satisfies the existing `check` assertions but violates some part of the specification which the user has decided not to provide. However, for the purposes of assertion checking, we say that a program is *correct* w.r.t. the given assertions for given valid queries if all its assertions have been proved for all the states that may appear in the computation of the program with the given valid queries.

A consequence of our assumptions so far, is that the overall framework needs to deal throughout with *approximations* [2.7, 2.17, 2.31]. Thus, while the system can be complete with respect to decidable properties (e.g., certain type systems), it cannot be complete in general, and the system may or may not be able to prove in general that a given assertion holds. The overall operation of the system will be sometimes imprecise but must always be *safe*. This means that all violations of assertions flagged by the preprocessor should indeed be violations, but perhaps there are assertions which the system cannot determine to hold or not. As already discussed in Chapter 1, this also means that we cannot in general reject a program because the preprocessor has not been able to prove that the complete specification holds.

Thus, and returning to our overall debugging process using the preprocessor (Figure 2.2), it may be the case that after some iterations of the checking/program correction cycle the preprocessor is not capable of detecting additional errors at compile-time nor to guarantee that the program is correct w.r.t. the existing `check` assertions. In such a situation the preprocessor allows the possibility of performing dynamic checking of assertions, i.e., introducing run-time tests into the program (indicated as `Program + RT tests` in Figure 2.2). Execution of the resulting program on the underlying CLP

[1] Note however that if `check` assertions exist for such parts of the program they are automatically checked.

system may then issue two kinds of error messages: those directly generated by the CLP system (`System RT error`) due to incorrect run-time calls to system predicates and those produced by the code added for run-time checking (`User RT error`) if some user-provided assertion is detected to be violated while executing the program.[2]

Our approach is strongly motivated by the availability of powerful and mature static analysers for (constraint) logic programs, generally based on abstract interpretation [2.17]. These analysers have proved quite effective in statically *inferring* a wide range of program properties accurately and efficiently, for realistic programs, and with little user input (see, e.g., [2.33, 2.38, 2.13, 2.26, 2.27, 2.36, 2.5, 2.6] and their references). Such properties can range from types and modes to determinacy, non-failure, computational cost, independence, or termination, to name a few. Traditionally, the results of static analyses have been applied primarily to program optimisation: parallelisation, partial evaluation, low-level code optimisation, etc. However, as we have seen, herein we will be applying static analysis in the context of *program development* (see, e.g., [2.2, 2.7, 2.31, 2.30]), and, in particular, in validation and error detection. This fits within a larger overall objective (achieved to a large extent in CiaoPP [2.30]), which is to combine both program optimisation and debugging into an integrated tool which uses multiple program analyses and (abstract) program specialisation [2.41, 2.40] as the two main underlying techniques.

2.1.2 Chapter Outline

In the following sections, we describe the preprocessor, mainly by means of examples, and discuss some technical issues of the correctness checking it performs. We present the overall framework of the preprocessor (i.e., we give some insight into the blank box of Figure 2.2) in Section 2.2. In Section 2.3 the techniques used for assertion checking both at compile-time and at run-time are discussed. The ideas presented are summarised and exemplified in Section 2.6 following a running example. In Section 2.5 we present how to customise the preprocessor for a particular CLP system. Finally, Section 2.7 discusses some practical hints on the use of assertions for correctness checking in our system.

In the rest of the text, we will use examples written in both CHIP [2.16, 2.1] and Ciao[3] [2.4, 2.29] and output from the corresponding preprocessors, i.e., CHIPRE [2.9] and CiaoPP [2.8, 2.30]. We will be mixing explanations at a tutorial level with some more technical parts in an effort to provide material that is instructive for newcomers but also has enough detail for more expert readers.

[2] As we will see, the messages from system predicates can in fact be also made come from the assertions present in the system libraries or those used to describe the built-ins.

[3] The Ciao system is available at `http://www.clip.dia.fi.upm.es/Software/`.

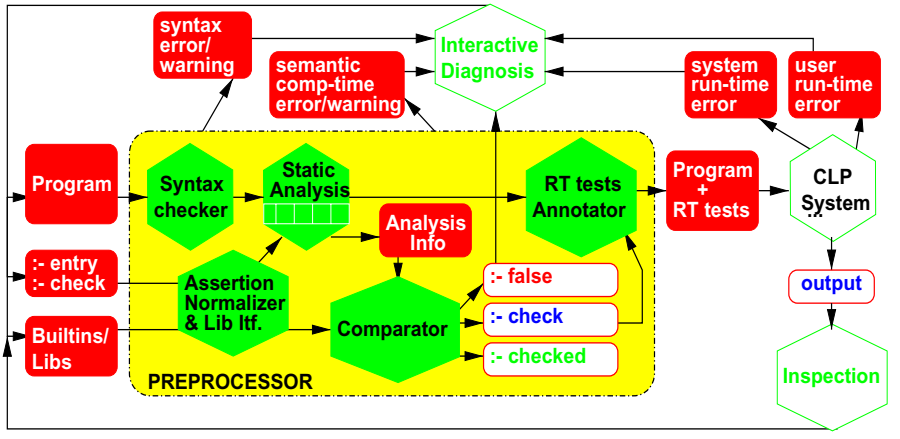


Fig. 2.3. Architecture of the Preprocessor

2.2 Architecture and Operation of the Preprocessor

The preprocessor is a complex system composed of several tools. Figure 2.3 depicts the overall architecture of the generic preprocessor. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools. It is a design objective of the preprocessor that most of such communication be performed also in terms of assertions. This has the advantage that at any point in the debugging process the information is easily readable by the user. In this section we provide an overall description of the different components of the preprocessor and give an overview of the kinds of bugs the preprocessor can automatically detect by means of examples. Further examples and details on some components will be given in other sections throughout the remainder of the chapter.

2.2.1 The Syntax Checker

As mentioned before, the preprocessor performs an extension of the syntax-level checking performed on programs prior to execution by traditional CLP systems. Though for simplicity only error messages were depicted in Figure 2.1, the preprocessor (as the CLP system) may issue error and/or warning messages (Figure 2.3).

Example 2.2.1. Consider the program in Figure 2.4, which contains a tentative version of a CHIP program for solving the *ship* scheduling problem, which is one of the typical benchmarks of CHIP. When preprocessing this program with CHIPRE, i.e., an implemented instance of the generic preprocessor for the CHIP system, the following messages are generated:

```
solve(Upper,Last,N,Dis,Mis,L,Sis):-
      length(Sis, N),
      Sis :: 0..Last,
      Limit :: 0..Upper,
      End :: 0..Last,
      set_precedences(L, Sis, Dis),
      cumulative(Sis, Dis, Mis, unused, unsed, Limit, End, unused),
      min_max(labeling(Sis), End).

labeling([]).
labeling([H|T]):-
      delete(X,[H|T],R,0,most_constrained),
      indomain(X),
      labeling(R).

set_precedences(L, Sis, Dis):-
      Array_starts=..[starts|Sis], % starts(S1,S2,S3,...)
      Array_durations=..[durations|Dis], % durations(D1,D2,D3,...)
      initialize_prec(L,Array_starts),
      set_pre_lp(l, array_starts, Array_durations).

set_pre_lp([],Array_starts).
set_pre_lp([After #>= Before|R], Array_starts, Array_durations):-
      arg(After, Array_starts, S2),
      arg(Before, Array_starts, S1),
      arh(Before, Array_durations, D1),
      S2 #>= S12 + D1,
      set_pre_lp(R, Array_starts, array_durations).

initialize_prec(_,_).
```

Fig. 2.4. A tentative *ship* program in CHIP

```
WARNING: (lns 34-35) predicate set_pre_lp/2:
                     has singleton variable(s) [Array_starts]
WARNING: (lns 35-42) predicate set_pre_lp/3:
                     already defined with arity 2
WARNING: (lns 35-42) predicate set_pre_lp/3:
                     has singleton variable(s) [S1,S12]
WARNING: (lns 35-42) predicate arh/3 undefined in source
```

The first message indicates that the variable `Array_starts` is a *singleton*, i.e., it only appears once in the clause. This often indicates that we have mistyped the name of a variable. However, in this case this does not correspond to a bug. `Array_starts` is a singleton because its value is not used anywhere else in the clause. The second message is issued because there are two predicates in the program with the same name but different arity. This often indicates that we have forgotten an argument in the head of a clause or we have added extra ones. In fact, we have forgotten the third argument in the first clause

of `set_pre_lp`. The third message marks both `S1` and `S12` as singletons. This is actually a bug as we have typed `S12` instead of `S1` in the second clause. The last message indicates that there is a call to a predicate which is not defined. This is because in the second clause for predicate `set_pre_lp` there is a call `arh(Before, Array_durations, D1),` where the name of the CHIP built-in predicate `arg` has been mistyped. However, unless we actually check that `arh` does not correspond to some user-defined predicate (by looking at the whole program) we cannot be sure that it corresponds to an undefined predicate.[4]

In addition to warning messages like the ones seen in the example above, the preprocessor also issues warning messages when the clauses which define a predicate are discontiguous. This may indicate that we have mistyped the name of the predicate in a clause or that an old clause for the predicate which should have been deleted is still in the program text. Some CLP systems already perform some of the checks discussed, mainly the singleton variable check.

As mentioned before, it is usually easy to make small (style) modifications to a program in order to avoid generating warning messages which do not correspond to bugs. Singleton variables can be replaced by *anonymous variables*, whose name start with an underscore. Predicates with the same name but different arities can be renamed to avoid name coincidence. Discontiguous clauses can always be put together. However, if we prefer not to modify the program, we can also instruct the preprocessor not to issue warning messages by setting the corresponding flag off.

Example 2.2.2. The following directive tells the preprocessor not to issue any warning message for predicates with the same name and different arity (until the flag is set to on):

```
:- set_prolog_flag(multi_arity_warnings,off).
```

Guided by the warning messages discussed above, we now replace the definition of predicate `set_pre_lp` by the following one, for which no syntactic warning message is issued anymore:

```
set_pre_lp([],_,_).
set_pre_lp([After#>=Before|R],Array_starts,Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arg(Before, Array_durations, D1),
        S2 #>= S1 + D1,
        set_pre_lp(R, Array_starts, array_durations).
```

The extra syntactic checking done by the preprocessor is based on that performed by the Ciao modular compiler [2.11]. This checking is simple to

[4] A well designed module system is instrumental in the task of detecting undefined predicates, especially when performing separate compilation [2.12].

implement, efficient to perform, as it does not need any complex analysis, and very relevant in practice: it allows early bug detection and it does not require any additional input from the user.

2.2.2 The Static Analysers

Though syntactic checking is simple, it is also very limited. In fact, there are plenty of incorrect programs which syntactically look fine. In order to go beyond syntactic checking, we require *semantic checking*. This kind of checking involves "understanding" to some extent what a piece of code does. This task is performed at compile-time by so-called *static analysis*. Static analysis is capable of inferring some useful properties of the behaviour of a program without actually running it. By running the program on sample input data we can obtain very precise information on the behaviour of the program, but such information is not guaranteed to be correct for other input data. In contrast, information obtained by means of static analysis is indeed guaranteed to be correct for *any* input data. However, analysis has to approximate its results in order to ensure termination, and this causes analysis to lose some information.

Example 2.2.3. Consider the following toy program:

```
p(X,Y):- Y is 2*X + 1.
```

By executing the program with input value `X=0` we can conclude that on success of the program `Y=1`. Static analysis can conclude that, for example, on success of the program the following properties hold for argument `Y`: `integer(Y)` (using type analysis), `ground(Y)` (using mode analysis), and `odd(Y)` (using parity analysis) independently of the particular value of `X`.

Unfortunately, static analysis is a hard task. In fact, no existing commercial CLP system contains a full fledged static analyser.[5] Thus, they cannot perform effective semantic compile-time checking. However, several generic analysis engines, generally based on abstract interpretation [2.17], such as PLAI [2.38], GAIA [2.13] and the CLP($\mathcal{R}$) analyser [2.37], facilitate construction of static analysers for (C)LP. These generic engines have the *description domain* as parameter. Different domains give analysers which provide different kinds of information and degrees of accuracy.

Static analysis can be *local* or *global*. In local analysis different parts of the program can be treated independently from others, for example, by processing one clause at a time. In global analysis, the whole program has to be

[5] Some exceptions do exist in the academic world, notably the &-Prolog system (the predecessor of Ciao and the first LP system to include a global analyser –PLAI– to perform optimisation tasks), the Aquarius and PARMA systems (the first LP systems to perform low-level optimisations based on global analysis), and the latest version of the CLP(R) compiler.

taken into account as the information obtained when processing other clauses (predicates) is possibly needed for processing a given clause (predicate). Thus, global analysis usually requires several iterations of analysis over the program code. Also, care must be taken to ensure termination of analysis. As a result, local analysis is generally simpler and more efficient than global analysis, but also less accurate.

The success of compile-time checking greatly depends on the accuracy of static analysis. As mentioned in Chapter 1, if analysis is goal-dependent, the accuracy of analysis can be improved by providing accurate `entry` declarations.

Example 2.2.4. In the ship program, all initial queries to the program should be to the `solve` predicate. However, the compiler has no way to automatically determine this. Thus, if no `entry` assertions are given for the program then most general entry assertions, i.e., of the form '`:- entry p(X1,...,Xn) : true.`' have to be assumed for *all* predicates `p/n` in the program.[6] However, if some entry assertion(s) exist they are assumed to cover all possible initial calls to the program. Thus, even the simplest entry declaration which can be given for predicate `solve`, i.e., '`:- entry solve(A,B,C,D,E,F,G) : true.`' (which can also be written using some syntactic sugar as '`:- entry solve/7.`') is very useful for goal-dependent static analysis. Since it is the only `entry` assertion, the only calls to the rest of the predicates in the program are those generated during computations of `solve/7`. This allows analysis to start from the predicate `solve/7` only, instead of from all predicates, which can result in increased precision. However, analysis will still make no assumptions regarding the arguments of the calls to `solve/7`. This can be improved using a more accurate entry declaration such as the following:

```
:- entry solve/7 :
         int * int * int * list(int) * list(int) * list * term.
```

which is syntactic sugar for '`:- entry solve(A,B,C,D,E,F,G): (int(A),int(B), int(C),list(D,int),list(E,int),list(F),term(G)).`'. It gives the types of the input arguments, and describes more precisely the valid input data.[7]

2.2.3 Consistency of the Analysis Results

The results of static analysis are often good indicators of bugs, even if no assertion is given. This is because "strange" results often correspond to bugs.

[6] Another advantage of a strict module system such as that of Ciao [2.12] is that only exported predicates of a module can be subject to initial queries. Thus most general `entry` assertions need only be assumed for exported predicates.

[7] Note that since, as mentioned in Chapter 1, properties (and, therefore, types) are considered *instantiation properties* by default, the assertion above also specifies a *mode*: all arguments except the last two are required to be ground on calls. The last but one argument is only *required* to be instantiated to a list skeleton, while no constraint is placed on the last argument.

As is the case with the syntactic warnings presented before, these indicators should be taken with care (as they do not ensure any violation of assertions) and thus warning messages rather than error messages are produced. An important observation is that plenty of static analyses, such as modes and regular types, compute over-approximations of the success sets of predicates. Then, if such an over-approximation corresponds to the empty set then this implies that such predicate never succeeds. Thus, unless the predicate is dead-code,[8] this often indicates that the code for the predicate is erroneous since every call either fails finitely (or raises an error) or loops.

Example 2.2.5. When preprocessing the current version of our example program using *regular types* [2.44, 2.18, 2.26] (see also Chapter 4 for a detailed discussion on regular types) analysis we get the following messages:

```
WARNING: Literal set_precedences(L, Sis, Dis)
           at solve/7/1/5 does not succeed!
WARNING: Literal set_pre_lp(l, array_starts, Array_duration)
           at set_precedences/3/1/4 does not succeed!
```

The first warning message refers to a literal (in particular, the 5th literal in the 1st clause of `solve/7`) which calls the predicate `set_precedences/3`, whose success type is empty. Also, even if the success type of a predicate is not empty, i.e., there may be some calls which succeed, it is possible to detect that at a certain program point the given call to the predicate cannot succeed because the type of the particular call is incompatible with the success type of the predicate. This is the reason for the second warning message.[9] Note that the predicate `set_pre_lp/3` can only succeed if the value at the first argument is compatible with a list. However, the call `set_pre_lp(l, array_starts, Array_duration)` has the constant `l` at the first argument. This is actually a bug, as `l` should be `L`. Once we correct this bug, in subsequent preprocessing of the program both warning messages disappear. In fact, the first one was also a consequence of the same bug which propagated to the calling predicates of `set_precedences/3`.

2.2.4 The Assertion Normaliser

As seen in Chapter 1, the assertion language in addition to the "basic" syntax for assertions also has an "extended" syntax which can be seen as syntactic sugar for writing assertions. The role of this module is to convert assertions possibly written in the extended syntax into the basic assertion language.

[8] If analysis is goal-dependent and thus also computes an over-approximation of the calling states to the predicate, predicates which are dead-code can often be identified by having an over-approximation of the calling states which corresponds to the empty set.

[9] This kind of reasoning can only be done if the static analysis used infers properties which are *downwards closed*.

For example, compound assertions (see Chapter 1) are converted into basic ones prior to compile-time checking. This module is represented in Figure 2.3 by the hexagon labelled `Assertion Normaliser & Lib Itf`. This module is also in charge of generating and reading *interface* files for modules. Interface files contain assertions describing the predicates exported by the module. This allows correctly analysing and checking programs composed of several modules without having to preprocess the auxiliary modules over and over again [2.42]. I.e., interface files allow modular analysis and assertion checking.

2.2.5 The Assertion Comparator

A simple (but tedious) possibility in order to use the information obtained by static analysis for detecting correctness problems at compile-time is to visually inspect the analysis results: unexpected results often indicate correctness problems. A more attractive alternative is to automatically compare the analysis results with our expectations, given as assertions.

As depicted in Figure 2.3, this is done in the preprocessor by the tool named `Comparator`. The result might be that the assertion is validated or that it is proved not to hold. In the first case the corresponding assertions are rewritten as `checked` assertions; in the second case *abstract* symptoms are detected, the corresponding assertions are rewritten as `false` assertions, and error messages are presented to the user. It is also possible that an assertion cannot be proved nor disproved. In this case some assertions remain in `check` status, and warning messages could be presented to the user to indicate this. In the case that errors are generated, diagnosis should be started. One option is to use the type-based diagnoser presented in Chapter 4 if the properties in the assertion are regular types, or other forms of abstract diagnosis [2.14, 2.15], in order to detect the cause of the error. Also, as we will see, the preprocessor does perform a certain amount of diagnosis, in the sense that it locates not only the assertion from which the the error or warning is generated but also, for example, the clause body literal that originates the call which is identified to be in error (see Section 2.2.7). Also, we believe it is not difficult to extend the preprocessor to perform more extensive diagnosis, using the quite detailed information on dependencies between program points that the analysers keep track of.

2.2.6 Assertions for System Predicates

A very important feature of the preprocessor, explained in detail in Section 2.5.1, is that the behaviour of system predicates (not only of user predicates) is given in terms of assertions. By *system predicates* we denote both built-ins and library predicates provided by the programming system. As many as possible of these system predicates should be described using assertions. Such assertions for system predicates, which are depicted in Figure 2.3

as `BuiltIn/Libs`, are in principle meant to be written by system developers when generating a particular instance of the preprocessor for a given CLP system. For example, the assertions describing the system predicates in Ciao have been written by the authors of this chapter and other Ciao developers and are part of the Ciao system libraries, whereas the assertions describing the built-in predicates in CHIP have been written at COSYTEC (i.e., the CHIP developers).

The existence of assertions which describe system predicates is beneficial for at least two reasons. One which is seemingly simple but quite relevant in practice is the possibility of automatic generation of documentation (primarily reference manuals) directly from the assertions [2.28]. In fact, the documentation of the preprocessor itself [2.9, 2.8] is generated this way. Another one is that though system predicates are in principle considered correct under the assumption that they are called with valid input data, it is still of use to check that they are indeed called with valid input data. In fact, existing CLP systems perform this checking at run-time. The existence of such assertions allows checking the calls to system predicates at compile-time in addition to run-time in CLP systems which originally do not perform compile-time checking. This may allow automatically detecting many bugs at compile-time without any user-provided assertions. The only burden on the user is (1) to optionally provide one or more `entry` assertions which describe the valid queries and (2) to wait the time required by the preprocessor in order to both perform static analysis and compare the analysis results with the expected calls to system predicates.

Example 2.2.6. Consider the current version of the *ship* program, and assume that the only existing entry declaration is '`:- entry solve/7.`'. When preprocessing the program the following messages are issued:

```
ERROR: Builtin predicate
        cumulative(Sis,Dis,Mis,unused,unsed,Limit,End,unused)
        at solve/7/1/6 is not called as expected (argument 5):
         Called:    ^unsed
         Expected:  intlist_or_unused

ERROR: Builtin predicate arg(After,Array_starts,S2)
        at set_pre_lp/3/2/1 is not called as expected (argument 2):
         Called:    ^array_starts
         Expected:  struct
```

In error messages, the marker `^` is used to distinguish constants (terms) from regular types (which represent sets of terms). By default, values represent regular types. However, if they are marked with `^` they represent constants. In our example, `intlist_or_unused` is a type since it is not marked with `^` whereas `^unsed` is a constant.[10]

[10] Though it is always possible to define a regular type which contains a single constant such as `unsed`, we introduce the marker `^` ("quote") to improve the rea-

The first message is due to the fact that the constant `unused` has been mistakingly typed as `unsed` in the fifth argument of the call to the CHIP built-in predicate `cumulative/8`. As indicated in the error message, this predicate requires the fifth argument to be of type `intlist_or_unused` which was defined when writing assertions for the system predicates in CHIP and which indicates that such argument must be either the constant `unused` or a list of integers. The exact definition of this type can be found in Section 2.5.1.

In the second message we have detected that we call the CHIP built-in predicate `arg/3` with the second argument bound to `array_starts` which is a constant (as indicated by the marker `^`) and thus of arity zero. This is incompatible with the expected call type `struct`, i.e., a structure with arity strictly greater than zero. In the current version of CHIP, this will generate a run-time error, whereas in other systems such as Ciao and SICStus, this call would fail but would not raise an error. Though we know the program is incorrect, the literal where the error is flagged, `arg(After, Array_starts, S2)` is apparently correct. We correct the first error and leave detection of the cause for the second error for later.

The different behaviour of seemingly identical built-in predicates (such as `arg/3` in the example above) in different systems further emphasises the benefits of describing built-in predicates by means of assertions. They can be used for easily customising static analysis for different systems, as assertions are easier to understand by naive users of the analysis than the hardwired internal representation used in ad-hoc analysers for a particular system.

Run-time checking of assertions describing system predicates presents some peculiarities. When the system predicates (of a particular CLP system) already perform the required run-time checks, the preprocessor does not introduce any extra code for run-time checking, even if this option has been selected (in contrast to what happens for user-provided assertions). However, if we design a CLP system from the start which is always going to be used in conjunction with the preprocessor a very interesting alternative exists: we can avoid introducing in the system predicates any code for checking the calls, as that can be done by the preprocessor. The advantage of this approach is that programs can be more efficient, as the preprocessor may prove that many of the run-time tests are redundant, and thus not introduce run-time tests for them.

2.2.7 Assertions for User-Defined Predicates

Up to now we have seen that the preprocessor is capable of issuing a good number of error and warning messages even if the user does not provide any `check` assertions. We believe that this is very important in practice. However,

dability and conciseness of the messages. Note that defining such type explicitly instead would require inventing a new name for it and providing the definition of the type together with the error message.

adding assertions to programs can be useful for several reasons. One is that they allow further semantic checking of the programs, since the assertions provided encode a partial specification of the user's intention, against which the analysis results can be compared. Another one is that they also allow a form of diagnosis of the error symptoms detected, so that in some cases it is possible to automatically locate the program construct responsible for the error. This is important since it is often the case that the preprocessor issues an error message at a program point which is actually far from the cause of the error. This happens, for example, when a predicate is called in the wrong way. Thus, one possibility is to visually inspect the program in order to detect which is (are) the wrong call(s) to the predicate among all the existing ones in the program. This does not seem like a good idea if the program being debugged is large and such predicate is used in several places. Thus, a better alternative is to add to the program an assertion on the calls to such predicate. This assertion can then be used by the preprocessor in order to automatically detect the wrong call(s) to the predicate.

Example 2.2.7. Consider again the pending error message from the previous iteration over the ship program. We know that the program is incorrect because (global) type analysis tells us that the variable `Array_starts` will be bound at run-time to the constant `array_starts`. However, by just looking at the definition of predicate `set_pre_lp` it is not clear where this constant comes from. This is because the cause of this problem is not in the definition of `set_pre_lp` but rather in that the predicate is being used incorrectly (i.e., its precondition is violated). We thus introduce the following `calls` assertion:

```
:- calls set_pre_lp(A,B,C): (struct(B),struct(C)).
```

In this assertion we require that both the second and third parameters of the predicate, i.e., `B` and `C` are structures with arity greater than zero, since in the program we are going to access the arguments in the structure of `B` and `C` with the built-in predicate `arg/3`.

The next time our ship program is preprocessed, having added the `calls` assertion, besides the pending error message of Example 2.2.6, we also get the following one:

```
ERROR: false assertion at set_precedences/3/1/4
       unexpected call (argument 2):
         Called:   ^array_starts
         Expected: struct
```

This message tells us the exact location of the bug, the fourth literal of the first clause for predicate `set_precedences/3`. This is because we have typed the constant `array_starts` instead of the variable `Array_starts` in such literal.

Thus, as shown in the example above, user-provided `check` assertions may help in locating the actual cause for an error. Also, as already mentioned,

and maybe more obvious, user-provided assertions may allow detecting errors which are not easy to detect automatically otherwise.

Example 2.2.8. After correcting the bug located in the previous example, preprocessing the program once again produces the following error message:

```
ERROR: false assertion at set_pre_lp/3/2/5
       unexpected call (argument 3):
         Called:   ^array_durations
         Expected: struct
```

which would not be automatically detected by the preprocessor without user-provided assertions. The obvious correction is to replace `array_durations` in the recursive call to `set_pre_lp` in its second clause with `Array_durations`. After correcting this bug, preprocessing the program with the given assertions does not generate any more messages.

Finally, and as already discussed in Section 2.1.1, if some part of an assertion for a user-defined predicate has not been proved nor disproved during compile-time checking, it can be checked at run-time in the classical way, i.e., run-time tests are added to the program which encode in some way the given assertions. Introducing run-time tests by hand into a program is a tedious task and may introduce additional bugs in the program. In the preprocessor, this is performed by the `RT tests Annotator` tool in Figure 2.3. How run-time checks are introduced is discussed in detail in Section 2.4.

2.3 Compile-Time Checking

Compile-time checking of assertions is conceptually more powerful than run-time checking. However, it is also more complex. Since the results of compile-time checking are valid for *any* query which satisfies the existing `entry` declarations, compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. Unfortunately, static analysers are complex to build and also some properties are very difficult to prove without actually running the program. On the other hand, though run-time checking is simpler to implement than compile-time checking, it also has important drawbacks. First, it requires test cases, i.e., sample input data, which typically have an incomplete *coverage* of the program execution paths. Therefore, run-time checking cannot be used in general for proving that a program is correct with respect to an assertion, i.e., that the assertion is `checked`, as this would require testing the program with all possible input values, which is in general unrealistic. Second, run-time checking clearly introduces overhead into program execution. Thus, it is important that we have the possibility of turning it on and off, as is the case in the preprocessor.

We now show informally how the actual checking of the assertions at compile-time is performed by means of an example. Then, we briefly discuss on the technique used in the preprocessor for "reducing" (i.e., validating and detecting violations of) assertions. Precise details on how to reduce assertions at compile-time can be found in [2.39].

Example 2.3.1. Assume that we have the following declarations of properties and user-provided assertions:

```
:- shfr prop ground/1.
:- shfr prop var/1.
:- regtype prop list/2.
list([],_P).
list([X|Xs],P):- P(X), list(Xs,P).
:- non_failure cprop does_not_fail/1.
:- cprop terminates/1.

:- check success p(X,Y) : ground(X) =>  ground(Y).
:- check success p(X,Y) =>  (list(X,int), list(Y,int)).
:- check comp    p(X,Y) :
                (list(X,int), var(Y)) + (does_not_fail,terminates).
```

which declare `ground/1`, `var/1`, and `list/2` as properties of execution states and `does_not_fail/1` and `terminates/1` as properties of computations (see Chapter 1). In addition, the declarations inform the preprocessor that properties `ground/1` and `var/1` can be treated using the inference system `shfr`, property `list/2` using `regtypes`, and `does_not_fail/1` using the `non_failure` inference system.[11]

As already mentioned in Chapter 1, assertion checking can be seen as computing the truth value of assertions by composing the value $eval(AF, \theta, P, IS)$ of the atomic properties AF at the corresponding stores θ reachable during execution. In the case of compile-time checking we must consider all possible stores reachable from any valid query. The abstract interpretation-based inference systems `shfr` and `regtype` compute a description (abstract substitution) for the calls and success states of each predicate (in fact they also do so for every program point). In compile-time checking we consider an *abstract evaluation* function $eval_\alpha(AF, \lambda, P, IS)$ in which the concrete store θ has been replaced by an abstract description λ. We denote by $\gamma(\lambda)$ the set of stores which a description λ represents. Correctness of abstract interpretation guarantees that $\gamma(\lambda)$ is a safe approximation of the set of all possible stores reached from valid initial queries, i.e., all such states are in $\gamma(\lambda)$.

[11] It is worth mentioning that since the properties shown in the example above are quite standard it is good practice to have their declarations (and possibly also their definitions) in a library module, so that users can simply include the module in their code rather than having to write the declarations from scratch for every new program. Also, in each particular instantiation of the preprocessor a set of such library modules should exist which are adapted to the analyses available.

Example 2.3.2. After performing static analysis of the program (whose text we do not show as the discussion is independent of it) using `shfr` and `regtypes` we obtain a description of the calls and success states for predicate `p/2`. For readability, we now show the results of such analyses in terms of assertions:

```
:- true success p(X,Y):(ground(X),var(Y)) => (ground(X),ground(Y)).
:- true success p(X,Y):(list(X,int),term(Y))=>(list(X,int),int(Y)).
```

We denote by $\lambda_c(\texttt{p/2})$ and $\lambda_s(\texttt{p/2})$ the description of the calling and success states, respectively, of `p/2`. In our example, the static inference system `shfr` allows us to conclude that the evaluation of the three atomic properties $eval_\alpha(\texttt{ground(X)}, \lambda_c(\texttt{p/2}), P, \texttt{shfr})$, $eval_\alpha(\texttt{ground(Y)}, \lambda_s(\texttt{p/2}), P, \texttt{shfr})$, and $eval_\alpha(\texttt{var(Y)}, \lambda_c(\texttt{p/2}), P, \texttt{shfr})$ take the value *true*. Additionally, the `regtypes` analysis determines that on success of `p/2`, i.e., in $\lambda_s(\texttt{p/2})$, the type of argument `Y` is `int`, which is a predefined type in `regtypes`. This type is incompatible with `Y` being a list of integers, which is what was expected. Thus, $eval_\alpha(\texttt{list(Y,int)}, \lambda_c(\texttt{p/2}), P, \texttt{regtypes})$ takes the value $false$. The implementation of $eval_\alpha$ in the preprocessor is discussed below and is based on the notion of *abstract executability* [2.41, 2.40].

Regarding the `non_failure` inference system, we assume it is implemented (as in [2.19]) as a program analysis which uses the results of the `shfr` and `regtypes` analyses for approximating the calling patterns to each predicate, and then infers whether the program predicates with the given calling patterns might fail or not based on whether the type is recursively "covered".

Example 2.3.3. Assume that `non_failure` analysis concludes that `p/2` does not fail for its calling pattern `p(X,Y):(list(X,int),var(Y))`. This can be indicated by an assertion of the form:

```
:- true comp p(X,Y) : (list(X,int), var(Y)) + does_not_fail.
```

and thus $eval_\alpha(\texttt{does\_not\_fail(p(X,Y))}, \lambda_c(\texttt{p/2}), P, \texttt{non\_failure})$ is guaranteed to take the value *true*.

Assume also that we do not have any static inference system which can decide whether `terminates(p(X,Y))` with calling pattern `(list(X,int),var(Y))` holds or not. Thus, $eval_\alpha$ for this property has to take the value "don't know".[12]

The next step consists of composing and simplifying the truth value of each logic formula from the truth value computed by $eval_\alpha$.

Example 2.3.4. After composing the results of evaluating each atomic property in the assertion formulae we obtain:

[12] Note that this could also happen even if a termination analysis exists in the system. Termination analyses may not able in general to determine whether a given computation terminates or not.

```
:- check success p(X,Y) : true =>  true.
:- check success p(X,Y) =>  (true, false).
:- check comp    p(X,Y) : (true, true) +   (true, terminates).
```

We can now apply typical simplification of logical expressions and obtain:

```
:- check success p(X,Y) : true =>  true.
:- check success p(X,Y) =>  false.
:- check comp    p(X,Y) : true +   terminates.
```

The third and last step is to obtain, if possible, the truth value of the assertion as a whole. As assertion takes the value *true*, i.e., it is validated if either its precondition (more formally, the app_A formula of Chapter 1) takes the value *false* (i.e., the assertion is never applicable) or if its postcondition (more formally, the sat_A formula of Chapter 1) takes the value *true*. The postcondition of the first assertion of our example takes the value *true*. Thus, there is no need to consider such an assertion in run-time checking, and we can rewrite it with the tag `checked`. An assertion is violated if its precondition takes the value *true* and its postcondition takes the value *false*.[13] This happens to the second assertion in our example. Thus, we can rewrite it with the tag `false`. Whenever an assertion is detected to be false at compile-time, in addition to being rewritten with the `false` tag, the preprocessor also issues an error message. This allows the user to be aware of an incorrectness problem without looking at the assertions obtained by compile-time checking.

If it is not possible to modify the tag of an assertion, then such assertion is left as a `check` assertion, for which run-time checks might be generated. However, as the assertion may have been simplified, this allows reducing the number of properties which have to be checked at run-time.

Example 2.3.5. The final result of compile-time checking of assertions is:

```
:- checked success p(X,Y) =>  (ground(X),ground(Y)).
:- false   success p(X,Y) =>  (list(X,int), list(Y,int)).
:- check   comp    p(X,Y) +   terminates.
```

where the third assertion still has the tag `check` since it is not guaranteed to hold nor to be violated. Note also that the two assertions whose tag has changed appear as in the original version rather than the simplified one. The preprocessor does so as we believe it is more informative.

[13] There is a caveat in this case due to the use of over-approximations in program analysis. It may be the case that a compile-time error is issued which does not occur at run-time for any valid input data. This is because though any activation of the predicate would be erroneous, it may also be the case that the predicate is never reached in any valid execution but analysis is not able to notice this. However, we believe that such situations do not happen so often and also the error flagged is actually an error of the program code. Though it can never show up in the current program it could do so if the erroneous part of the program is used in another context (in which it is actually used) in another program.

One important consideration about compile-time checking is that, for a given analysis, not all properties are equally simple to reduce at compile-time to either *true* or *false*. Unless otherwise stated, we assume that, as is usually the case, analysis computes over-approximations (but it is straightforward to develop dual solutions for under-approximations). We first study the case of reducing a property to *true* and then the case for reducing it to *false*.

A declaration of the form '`:- InfSys prop Prop/n.`' implicitly states that there is some abstract description λ_{TS}(`Prop/n`) computable by the static analysis `InfSys` such that for all stores θ in $\gamma(\lambda_{TS}$(`Prop/n`)) the property `Prop/n` holds. In such case, if at the corresponding context, analysis computes a description λ such that $\gamma(\lambda) \subseteq \gamma(\lambda_{TS}$(`Prop/n`)) then the property is clearly guaranteed to hold for any store in $\gamma(\lambda)$. This process corresponds to abstractly executing the property to the value *true* [2.41, 2.40].

Another way of proving that a property holds may be provided by the existence of a declaration of the form '`:- proves Prop(X) : Prop2(X).`' (see Chapter 1), which indicates that in order to prove `Prop(X)` it is sufficient to prove `Prop2(X)`. Note that these `proves` declarations may be chained to any length. If we reach some `PropN(X)` for which a declaration '`:- InfSys prop PropN.`' exists, then we can try to prove it as described above.

Regarding reducing a property to *false*, analyses based on over-approximations are not so appropriate. Assume that the result of analysis is a description λ which is an over-approximation of the set of stores which may occur during the execution of a given program construct. The fact that there are elements in $\gamma(\lambda)$ in which a property AF does not hold does not guarantee that AF actually does not hold during execution. It is possible that such elements correspond to stores approximated by $\gamma(\lambda)$ but which do not actually occur during execution of the given program construct. They might have been introduced due to the loss of precision of the analysis. In any case, if the property AF does not hold in *any* of the elements of $\gamma(\lambda)$ then it does not hold in any of the actual stores either. This is a sufficient condition for the property to be false.

Despite the discussion above, and as we will see in the coming example, it is important to mention that it is possible to reduce a property `Prop/n` to *false* using an inference system `InfSys` without the need of any assertion of the form '`:- InfSys prop Prop/n`' which declares `Prop/n` as *directly provable* in `InfSys`, and even if no `disproves` assertion exists for `Prop/n`. This is very convenient, since in order to add an assertion '`:- InfSys prop Prop/n`' we need to know a suitable λ_{TS}(`Prop/n`) and this requires a good understanding of both the inference system `InfSys` and the property under consideration `Prop/n`. As a result of being able to reduce very general properties to *false*, the preprocessor can detect incorrectness problems at compile-time even if the properties used in the assertions have not been written with any static inference system in mind.

Example 2.3.6. Consider a property `sorted_num_list` to check for sorted lists of numbers (defined as in Section 2.6), and mode and regular type analyses. Although the property cannot be proved with only (over approximations) of mode and regular type information, it may be possible to prove that it does *not* hold (an example of how properties which are not provable of any analysis can also be useful for detecting bugs at compile-time). While the regular type analysis cannot capture perfectly the property `sorted_num_list(T)`, it can still approximate it (by analysing the definition) as `list(T,num)`. If type analysis for the program generates a type for `T` which is incompatible with `list(T,num)`, then a definite error symptom is detected.

Similarly to the case of reducing properties to *true*, a declaration of the form '`:- disproves Prop(X) : Prop2(X).`' can be used to disprove `Prop(X)` if we can prove `Prop2(X)`. Again, we can also have a chain of `proves` declarations which may allow proving `Prop2(X)` and thus disproving `Prop(X)`. Further discussion on how to deal with the different kinds of properties and the impact of approximations can be found in [2.7, 2.39].

Properties that are directly provable in an inference system `InfSys` are the main targets for abstract execution when using `InfSys` as static inference system, since it is often possible to either prove them or disprove them, provided that `InfSys` is accurate enough. This is the case for the properties `ground`, `list(int)` and `does_not_fail` in the example above. Note that, if the analysis `InfSys` is *precise* (in the sense that the abstract operations do not lose information beyond the abstraction implied by the abstraction function used [2.17]) and, obviously, terminates, then there are properties which can be reduced to either *true* or $false$ in all cases, i.e., for which the analysis is a complete inference system. In fact, in systems conceived for compile-time checking only, the properties admitted in assertions are usually restricted to those which are (almost) *perfect observables* of the analysis available, i.e., they can (almost) always be proven or disproven. This is not our case, and nevertheless, we have seen how interesting properties –which are not perfect observables– can still be used to detect incorrectness symptoms.

Though the process just shown simplifies predicate assertions using the information static analysis has inferred for the global (call and) success states for the predicate, in order to perform compile-time checking of program-point assertions the inference system needs to be able to compute information at each program point. In fact, both `shfr` and `regtypes` do so. It is also important to mention that it is also very useful to perform compile-time checking of predicate assertions at each program point. The interest in doing this is because many different calls to a predicate may exist in the program and it is likely that only some of them are incorrect. Thus, as some analyses summarise all possible calls in one description (by losing precision), it is not possible with these analyses to detect that an assertion is violated in *some* calls to the predicate. One way to solve this problem is to compute analysis information at each program point and then compare the analysis

information at each program point with the assertions for the corresponding predicate called at that particular program point.[14]

Be it at the predicate level or at the program-point level, the fact that an assertion remains in status `check` cannot be taken as an indicator of an incorrectness problem, as is the case in strong type systems, since it may be due to the lack of an static analysis for which the given property is provable, or to the loss of precision of the analyser. In any case, issuing a warning might be helpful, but (automatic) checking would have to be performed at run-time.

2.4 Run-Time Checking

The aim of run-time checking of assertions is to detect those assertions which are violated during the execution of the program for the data being explored. Such checking involves:

1. At each execution state it must be determined which are the assertions (if any) which *affect* the corresponding state together with the logic formulae which have to be evaluated at the corresponding store.
2. Actually evaluating the required atomic formulae AF, i.e. computing $eval(AF, \theta, P, IS)$.
3. Obtaining the truth value of the logic formulae as a whole.
4. Obtaining the truth value of the assertion as a whole. If the assertion is detected to be violated error messages should be issued.

As discussed earlier (Figure 2.2), run-time checking in the preprocessor is implemented as a program transformation which adds new code to the program as needed to perform the required checking during execution of the program. This program transformation, presented in Section 2.4.2 below, is in charge of solving item 1 above. As discussed in Chapter 1, item 2 above, i.e., evaluation of the atomic formulae, must be performed by a suitable inference system IS. We discuss how this evaluation is performed in our preprocessor in Section 2.4.1, under the assumption that we use the underlying CLP system as the inference system IS. This section also discusses how to obtain the truth value of the logic formulae as a whole, i.e., item 3 above. Finally, in Section 2.4.2 we present the overall program transformation and a series of predicates which are in charge of obtaining the truth value of each assertion as a whole and of issuing errors when an assertion is detected to be violated. The code of such predicates is independent of the particular program being checked and is stored in a library program named `rtchecks` which is loaded by the the transformed program (which requires such library in order to run

[14] Furthermore, since the analysis is multi-variant, a single program point may be reflected in analysis by several descriptions. Thus, it is in principle possible to perform assertion checking and diagnosis at a finer-grained level.

correctly). If the checking proves that an assertion is violated, a *concrete*[15] incorrectness symptom is detected and some kind of error message is given to the user. A procedure for localising the cause of the error, such as standard or declarative diagnosis can then be started. It is out of the scope of this chapter to discuss how program diagnosis should be performed. However, techniques such as declarative debugging [2.43, 2.3, 2.20, 2.21] (see Chapter 5) or more traditional interactive debuggers [2.10, 2.22] may be applied. Correctness of the transformation requires that the transformed program only produce an error message if the specification is in fact violated.

2.4.1 Evaluating Atomic Logic Formulae

As discussed in Chapter 1, one of the design decisions of our assertion language is that property predicates are defined in the source language. Given this, and although the assertion language design leaves the choice of implementation for the inference system open, it seems interesting to study the viability of using the underlying (C)LP system to perform the run-time checks, i.e., to study whether $eval(AF, \theta, P, IS)$ can be reduced to true or false with IS being the (C)LP system in which the program is running.

This introduces some simplifications in the problem: since IS is given, θ is the current store, and P is the program itself, then we can simplify $eval(AF, \theta, P, IS)$ to simply $eval(AF)$. Since AF is a predicate of the program, then the answer to this question could be obtained, at least at a first level of approximation, simply by calling AF directly and seeing whether it succeeds or fails. This raises at least two issues. First, the fact that the system will interleave calls to the code which defines each property predicate with the execution of the program imposes in practice some limitations on such code. Essentially, we would like the behaviour of the program not to change in a fundamental way independently of whether the program contains run-time checks or not. The second issue is whether we can use the result of the execution of AF in the current store as the value of $eval(AF)$.

Regarding the first issue above, while we can tolerate a degradation in execution speed, turning on run-time checking should not introduce non-termination in a program which otherwise terminates. Also, checking a property should not change the answers computed by the program or produce unexpected side-effects (such as new program errors, other than those from assertion violations). In light of this, a reasonable set of requirements on the definitions of the predicate properties is the following:

1. The execution of the code which defines the property predicate should terminate, and it should do so with success or failure, not error, *for any possible input.*

[15] As opposed to *abstract* incorrectness symptoms, which are the ones detected by compile-time checking.

2. The code defining a property should not perform input/output, add or delete clauses, etc., which may interfere with the program behaviour.
3. The execution of a property should not further instantiate its arguments or add new constraints, it may not change the store θ seen by the subsequent part of the execution.

In practice, these conditions can be relaxed somewhat in order to simplify the task of writing property predicates. In particular, and as we will see, the program transformation that we present below guarantees requirement 3, i.e., run-time checking is guaranteed not to introduce any undesired constraints, independently of how the property predicates are written. Thus, condition 3 can be ignored in practice. Also, some basic checks on the code defining property predicates are enough in most cases for requirement 2, i.e., the system can easily be made to reject a property predicate which does not satisfy condition 2. However, the user is responsible for guaranteeing termination of the execution of the properties defined. To this end, our system assumes that property predicates declared with `prop` (i.e., those of the execution states) will always terminate for any possible calling state.

We now turn to the second issue above, i.e., whether the result of executing AF in the current store can be used as the value of $eval(AF)$. Recall that the assertion language (Chapter 1) specifies that the checking of properties about execution states has to guarantee that they are treated as instantiation checks, unless they are qualified by `compat`, in which case they should be treated as compatibility checks. The fact that the assertion language allows writing the definitions of properties in such a way that they can be used both in instantiation and compatibility checks, imposes the need for some machinery beyond simply calling AF. It turns out that performing instantiation checks corresponds to performing what is referred in (C)LP as *entailment* and *disentailment* tests [2.32, 2.39]. A constraint is *entailed* by a constraint store if the constraint is implied by the store. Conversely, a constraint is *disentailed* by a constraint store if it is inconsistent with the store. We extend the notion of entailment and disentailment, originally defined on constraints, to property predicates. We say that a property predicate is entailed by a constraint store if its execution succeeds without affecting the store, i.e., any constraint added to the store only affects variables which are local to the definition of the predicate. Also, we say that a property predicate is disentailed by a constraint store if its execution fails. An instantiation property is true if it is entailed, and false if it is disentailed.

Given these definitions, the question then is how can entailment and disentailment tests be performed by evaluation of the property formulae and how the characteristics of the CLP system might affect this evaluation. Figure 2.5 lists definitions for `entailed/1` or `disentailed/1` which achieve this task, i.e., properties are treated as instantiation checks unless they are marked as `compat`, in which case they are treated as compatibility checks, and also make sure that checking does not affect program execution by posing any additional constraints on the store.

```
disentailed((LogForm,LogForms)):- !,
        ( disentailed(LogForm); disentailed(LogForms) ).
disentailed((LogForm;LogForms)):- !,
        disentailed(LogForm), disentailed(LogForms).
disentailed(compat(LogForm)):- !,
        system_dependent_incompatible(LogForm).
disentailed(AtomForm):-
        \+ \+ system_dependent_disentailed(AtomForm).
disentailed(AtomForm):-
        disproves(AtomForm,SufficientCond),
        entailed(SufficientCond).

entailed((LogForm,LogForms)):- !,
        entailed(LogForm), entailed(LogForms).
entailed((LogForm;LogForms)):- !,
        ( entailed(LogForm); entailed(LogForms) ).
entailed(compat(LogForm)):- !,
        system_dependent_compatible(LogForm).
entailed(AtomForm):-
        \+ \+ system_dependent_entailed(AtomForm).
entailed(AtomForm):-
        proves(AtomForm,SufficientCond),
        entailed(SufficientCond).
```

Fig. 2.5. Entailment and Disentailment of Logic Formulae

The implementation of Figure 2.5 also takes care of an additional issue: guaranteeing that the possible incompleteness of the constraint solver does not affect correctness of the dynamic assertion checking. By incompleteness we mean the fact that the solver may fail to detect that some state (constraint store) is inconsistent when it is indeed inconsistent. This may happen because of constraints which the solver decides to delay, as they are hard to treat in the current state, and will be taken into account when more information is available. Note that a simple sufficient condition for a property to be disentailed is that the execution of the predicate defining it actually fails, which means that the constraints it imposes are inconsistent with the store. If the solver is incomplete, properties which are false may go undetected, and preconditions which in fact do not hold might be deemed to hold, which is obviously incorrect.

Moreover, finding a sufficient condition for entailment of a property is not as easy as with disentailment. In fact, many CLP systems do not even have an entailment test for constraints. Also, checking that the constraint store remains the same after the execution of a property predicate (i.e., that its execution has not added constraints to the store) may not be an easy task, as store comparison is an operation which is usually not available. Therefore, these checks have to be defined separately for each CLP system. I.e., system implementors should provide suitable definitions for the

predicates `system_dependent_entailed`, `system_dependent_disentailed`, `system_dependent_compatible`, and `system_dependent_incompatible`. Possible implementations for a particular constraint system are given in Section 2.5.2.

Also, note that there are properties for which no *accurate* definition can be written in the underlying language, and therefore it is difficult that executable definitions are given for them. For these properties, an *approximate* definition may be given, and this approximation should be correct in the usual sense that all errors flagged should be errors, but there may be errors that go undetected. This can be done in the assertion language with `proves` and `disproves` assertions. These assertions are translated during the transformation of the program into two tables of facts, `proves/2` and `disproves/2`, which are then considered during run-time checking by the appropriate clauses of the predicates `entailed/1` and `disentailed/1`, as seen in the code presented above.

Finally, the situation is slightly different for properties of the computation, i.e., those property predicates declared with `cprop`. These predicates may have to reconstruct the computations of which they have to decide if the property they define holds or not. Since the necessary part of the computation required may be an infinite object, it is possible that the process of reconstructing such computation does not terminate, in which case the execution of the property predicate will not terminate either. Thus, we admit that the predicates for properties of the computation do not terminate, *provided* that the execution of the corresponding computation does not terminate either. Note that in this case run-time checking will not introduce non-termination into the program, in the sense that if the execution of the original program terminates then its execution with run-time checking will also terminate. It is also worth mentioning that the run-time translation presented is valid provided that the computation on which a property has to be checked does not perform side-effects (or the property is written in such a way that it captures calls to side-effects and avoids them). Otherwise, such side-effects may be performed more than once since after checking the assertion the computation is performed anyway.

2.4.2 A Program Transformation for Assertion Checking

This behaviour can be accomplished by a program transformation which, basically, precedes each call to a predicate involved in a predicate assertion with calls to the atomic properties in the precondition of the assertion, and postpones calls to the atomic properties in the postcondition until after success of the call.

We now provide a possible scheme for translation of a program with assertions into code which will perform run-time checking. Our aim herein is not to provide the best possible transformation (nor the best definition of auxiliary predicates used by it), but rather to present simple examples with

the objective of showing the feasibility of the implementation and hopefully clarifying the approach further.

Given a predicate p/n for which assertions have been given, the idea is to replace the definition of p/n so that whenever p/n is executed the assertions for it are checked and the actual computation originally performed by p/n is now performed by the new predicate p_new/n (where p_new stands for a fresh atom which is guaranteed not to coincide with other predicate names). I.e., given the definition of a predicate p/n as:

```
p(t11,...,t1n):- body_1.
          ...
p(tm1,...,tmn):- body_m.
```

it gets translated into:

```
p(X1,...,Xn):-
        check_assertions_and_execute ``p_new(X1,...,Xn)''.

p_new(t11,...,t1n):- body_1.
                ...
p_new(tm1,...,tmn):- body_m.
```

I.e., the definition of p_new/n corresponds to the definition of p/n in the original program and is independent of the assertions given for p/n. The checks present in the new definition of predicate p/n depend on the existing assertions for such predicate.

Success Assertions. A possible translation scheme for success assertions with a precondition is the following. Let A(p/n) represent the set of assertions for predicate p of arity n. Let RS be the set $\{(p(Y_1, ..., Y_n), Precond, Postcond)$ s.t. ‘:- success p(Y1,...,Yn) : $Precond$ => $Postcond$’ $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        prec(RS,p(X1,...,Xn),S),
        p_new(X1,...,Xn),
        postc(S,p(X1,...,Xn)).
```

where prec/3 collects the elements in RS s.t. *Precond* definitely holds at the calling state to p/n. A possible implementation of the prec/3 predicate is the following:

```
prec([],_Goal,[]).
prec([(Pred_desc, Precond, Postcond)|RS],Goal,NRS):-
        test_entailed(Precond,Pred_desc,Goal),!,
        NRS = [(Pred_desc, Precond, Postcond)|MoreRS],
        prec(RS,Goal,MoreRS).
prec([_|RS],Goal,NRS):-
        prec(RS,Goal,NRS).

test_entailed(LogForm,Pred_desc,Goal):-
        \+(\+((Pred_desc=Goal, entailed(LogForm)))).
```

note that those assertions whose precondition cannot be guaranteed to hold are directly discarded.

Finally, `postc/2` checks whether the *Postcond* of each assertion collected by `prec/3` holds or not. If it does not hold then the corresponding assertion is definitely violated and usually an error message will be given to the user (actually, in CiaoPP an exception is raised), and optionally, computation halted. If either they hold or they cannot be guaranteed not to hold computation will generally continue as usual. A possible implementation follows:[16]

```
postc([],_).
postc([(Pred_desc,Precond , Postcond)|_Cs],Goal):-
        Pred_desc = Goal,
        disentailed(Postcond),
        write('ERROR: for Goal '),
        write(Pred_desc), nl,
        write('with Precondition '),
        write(Precond), nl,
        write(' holds but Postcondition '),
        write(Postcond),
        write(' does not.'),nl,
        fail.
postc([_|Cs],Goal):-
        postc(Cs,Goal).
```

note that the call `p(X1,...,Xn)` is passed as an argument to both `prec/3` and `postc/2`. However, such call is not executed but rather it is used to pass the values of the arguments `X1,...,Xn` just before and after executing `p_new(X1,...,Xn)`. Something similar happens in the `calls/2` predicate introduced below.

Calls Assertions. A possible translation scheme for `calls` assertions is the following. As before, let A(`p/n`) be the set of assertions for predicate `p` of arity `n`. Let `C` be the set $\{(p(Y_1,...,Y_n), Precond)$ s.t. '`:- calls p(Y1,...,Yn)` : *Precond*' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        calls(C,p(X1,...,Xn)),
        p_new(X1,...,Xn).
```

where `calls/2` checks whether the preconditions for the predicate hold or not. If it detects that some *Precond* in `C` does not hold for some call to the predicate then the corresponding assertion is violated. A possible implementation of the `calls/2` predicate follows:

```
calls([],_Goal).
calls([(Pred_desc, Precond)|_Calls],Goal):-
        Pred_desc = Goal,
        disentailed(Precond),
```

[16] In the code we use for simplicity calls to `write/1`. However in the actual implementation this is better implemented by raising exceptions (using `throw`).

```
        write('ERROR: undefined call for predicate '),
        write(Goal), nl,
        fail.
calls([_|Calls],Goal):-
        calls(Calls,Goal).
```

Comp Assertions. A possible translation scheme for `comp` assertions is the following. Let `RC` be the set $\{(p(Y_1,...,Y_n), Precond, Comp\_prop)$ s.t. '`:- comp p(Y1,...,Yn)` : $Precond$ + $Comp\_prop$' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        check_comp(RC1,p(X1,...,Xn)),
        p_new(X1,...,Xn).
```

where in `RC1` we have added to each property of the computation in `RC` the goal `p_new(X1,...,Xn)` as the first argument.

Example 2.4.1. Given `RC=`{`(p(Y1,...,Yn), ground(Y1),deterministic)`} then `RC1= [(p(Y1,...,Yn), ground(Y1), deterministic(p_new(X1,...,Xn)))]`.

The `check_comp` predicate aims at checking $Comp\_prop$ for those `comp` assertions whose precondition holds. If $Comp\_prop$ is guaranteed not to hold then the corresponding assertion is definitely violated. A possible implementation follows:

```
check_comp([],_Goal).
check_comp([(Pred_desc,  Precond, CompProp)|_RC],Goal):-
        test_entailed(Precond,Pred_desc,Goal),
        Pred_desc = Goal,
        system_dependent_incompatible(CompProp),
        write('ERROR: for Goal '),
        write(Pred_desc), nl,
        write('with Precondition '),
        write(Precond), nl,
        write(' holds but CompProp '),
        remove_int_goal(CompProp,NCompProp),
        write(NCompProp),
        write(' does not.'), nl,
        fail.
check_comp([_|RC],Goal):-
        check_comp(RC,Goal).
```

Note that in this case in order to guarantee that $Comp\_Prop$ actually does not hold, `system_dependent_incompatible` rather than `disentailed` is used. This is because in contrast to properties of execution states, properties of the computation are not treated by default as instantiation properties since the computation itself may add constraints and the property can still be considered to hold. Such constraints will be eliminated from the store by forcing backtracking (see the definition of `system_dependent_incompatible` in Section 2.5.2).

Program-Point Assertions. Since they are part of the program, in order to execute a program which contains program-point assertions, a definition of the (meta) predicate `check/1` must be provided. A possible definition is the following:

```
check(LogForm):-
        disentailed(LogForm), !,
        write('ERROR: false program point assertion'),
        nl, write(LogForm),nl.
check(_).
```

where the definition of `disentailed` is given below. If run-time checking is not desired there are two possibilities, either the preprocessor can simply remove the program-point assertions from the program or, as already mentioned in Chapter 1, the alternative definition for `check/1` is used:

```
check(_).
```

2.5 Customising the Preprocessor for a CLP System: The CiaoPP and CHIPRE Tools

We have implemented the schema of Figure 2.3 as a *generic preprocessor*. This genericity means that different instances of the preprocessor for different CLP systems can be generated in a straightforward way. One reason for this is that within the DiSCiPl Project, several platforms exist (Ciao, CHIP, PrologIV, etc.) for which the preprocessor is of use. The preprocessor is parametric w.r.t. the set of system predicates, small syntax differences (mainly definition of operators) and the set of analysers available. The generic preprocessor is a stand-alone application written in Ciao Prolog, which is a public domain next generation logic programming system developed by the Clip group at UPM.

Currently, two different experimental debugging environments have been developed using this generic preprocessor:

- CiaoPP [2.8, 2.30], the Ciao system preprocessor, developed by the Clip group at UPM, and
- `CHIPRE` [2.9], a preprocessor for CHIP developed also by the Clip group at UPM, in collaboration with Pawel Pietrzak at Linköping University (adaptation of Gallagher's type analysis to the CLP(FD)language used by CHIP), and Helmut Simonis at COSYTEC (initial customisation assertions for supporting the CHIP built-in predicates –see Section 2.5.1– and development of a graphical user interface to the tool).

In order to port the preprocessor to another CLP system we basically require the following:

- The preprocessor should understand the syntactic and semantic particularities of the CLP system. Understanding the syntactic particularities generally amounts to the definition of a set of operators which are predefined in the language. Understanding the semantic particularities amounts to providing a description of the system predicates in the given CLP. This issue is further discussed in Section 2.5.1 below. This allows both performing accurate static analysis and checking calls to system predicates at compile-time.
- The CLP system should be able to read programs with assertions. This is needed since in order to exploit the full power of the preprocessor users should add assertions to their programs. This can be solved by adding to the CLP system a library program which contains the definitions of operators required for reading assertions. We provide such a library, called `assertions`, together with the distribution of the preprocessor.
- The CLP system should be able to run programs which have been subject to the run-time checking translation scheme. Such transformation includes in the program calls to predicates whose definition is also provided in a library program called `rt_checks`. Most of the code of this program is independent of the CLP system being used. However, depending on the system, some predicates have to be customised for the particular system being used. Much in the same way as assertions for built-in predicates, the definitions of such predicates are supposed to be provided by system developers rather than by users of the preprocessor. More details on this are presented in Section 2.5.2 below.

It is also important to mention that another way of customising the preprocessor is by integrating new analyses in addition to the existing ones. This can be done by defining new domains for analysis for the generic analysis engine i.e., PLAI [2.38], in the preprocessor. Due to space limitations, we do not go into the details on how to define new analysis domains here.

2.5.1 Describing Built-Ins Using Assertions

Assertions for built-in predicates should be provided by the system programmers to allow compile-time checking of the system predicates in the preprocessor (and also facilitate their run-time checking, as discussed before). The system predicate assertions must be available at the time of installing the preprocessor, so that it is configured for the particular language, and therefore, it understands the built-in predicates of the language. It is convenient to provide assertions that describe:

- Their success states. This is useful for improving the information obtained by static analysis. Note that many of the built-in predicates may be written in other languages, such as C. Therefore, it is not easy to automatically analyse their code with the analysers of the preprocessor, which are designed for CLP languages.

– Their required calling states. This is useful for performing compile-time checking of the calls to the system predicates. Unless this is done, checking of such calls has to be done at run-time.

The former can be described using `trust success` assertions, the latter with `check calls` assertions. They should define all and the only possible uses of a predicate.[17] If the preprocessor can determine that a system predicate is used in a way which is different from those described by the assertions for it, its behaviour is deemed to be unpredictable and a compile-time error is issued.

Example 2.5.1. Consider the `cumulative/8` built-in global constraint of CHIP. A call of the form:

```
cumulative(Starts,Durations,Resources,Ends,Surfaces,High,End,Int)
```

states, in its most simple form, that the cumulative use of a resource by all tasks with starts dates (in the list of finite domain ranges of) `Starts`, durations (in the list of ranges of) `Durations`, and resource use (in the list of ranges of) `Resources` is below the (range of) limit `High`. In this use of the constraint the arguments `Ends`, `Surfaces`, `End`, and `Int` are assigned the value `unused`. However, in its most general form, the cumulative constraint can be used with any of `Ends` and `End` assigned a list of ranges, `Int` a range, and/or `Surfaces` a list of integers.

To describe the behaviour of the cumulative constraint in terms of, e.g., regular types, one needs to use the type "list of finite domain variables" (i.e., `list(anyfd)`), the type "assigned value `unused`", as well as the types which describe the rest of the arguments. These types may be defined with the regular predicates:

```
:- regtype prop notused/1.
notused(unused).

:- regtype prop fd_or_unused/1.
fd_or_unused(unused).
fd_or_unused(X) :- anyfd(X).

:- regtype prop fdlist_or_unused/1.
fdlist_or_unused(unused).
fdlist_or_unused(X) :- list(X,anyfd).

:- regtype prop intlist_or_unused/1.
intlist_or_unused(unused).
intlist_or_unused(X) :- list(X,int).
```

so that the following assertion describes all the possible uses of the cumulative constraint:

[17] For this purpose `pred` assertions can be used, which are translated into one `check calls` and several `trust success` assertions.

```
:- trust pred cumulative/8
         : list(anyfd) * list(anyfd) * list(anyfd)
         * fdlist_or_unused * intlist_or_unused * anyfd
         * fd_or_unused * fdlist_or_unused
        => list(anyfd) * list(anyfd) * list(anyfd)
         * fdlist_or_unused * intlist_or_unused * anyfd
         * fd_or_unused * fdlist_or_unused .
```

However, it is also convenient to provide descriptions for particular uses of the predicates, since this may allow the analysers to improve accuracy. Detailed descriptions of different uses provide more information than a unique global description of all possible uses. In order to do this, one could have added success assertions for some (or all) of the possible uses of the predicate.

Example 2.5.2. The abovementioned use of the cumulative constraint (which corresponds to a typical use in manpower resource restricted scheduling), in which the arguments named above `Ends`, `End`, `Int`, and `Surfaces` are not used, is described by the (additional) assertion:

```
:- trust success cumulative/8
         : list(anyfd) * list(anyfd) * list(anyfd)
         * notused * notused * anyfd * notused * notused.
        => list(anyfd) * list(anyfd) * list(anyfd)
         * notused * notused * anyfd * notused * notused .
```

Though it may be argued that writing the assertions describing the system predicates is a tedious task, such task is not supposed to be performed by the users of the preprocessor, which can assume that such assertions are already present in the preprocessor. Also, this task is in any case definitely much simpler than writing an analyser from scratch or adapting an existing one written for another CLP system.

However, a disadvantage of describing system predicates in terms of assertions rather than in some lower-level description is a relative loss of efficiency. This is because, as is usual in software, the more general a piece of software is, the less efficient. In fact, software can often be optimised by *specialising* [2.35, 2.25] it w.r.t. some particular case. In our case, we would like our system to be at the same time easily portable to different systems and as efficient as if it had been designed with a particular system in mind. With this aim, we have implemented a program called `builtintables` which takes as input the assertions describing the system predicates and converts such information into the internal format of the analysers. This is conceptually equivalent to specialising an analyser to a particular set of system predicates. However, rather than using a general purpose specialiser, we follow the simpler to implement *generating extension approach* of partial evaluation in that the above mentioned program only knows how to specialise the analyser w.r.t. the assertions describing the system predicates. The `builtintables` program is only executed whenever the assertions for system predicates are modified.

2.5.2 System Dependent Code for Run-Time Checking

In order to perform run-time checking, a library `rtchecks` should be provided by the system. Most of the predicates in this library are independent of the underlying CLP system used, and possible implementations of such predicates have been given above. However, there are four predicates whose definition was not provided. This is because they depend on the particular CLP system being used.

The role of the system dependent predicates is basically to determine whether properties hold (are entailed) or not (are disentailed). Since individual properties may appear with the `compat` qualifier, we also need predicates for testing for compatibility and incompatibility.

We have chosen to define the system dependent predicates for the case of using Prolog as underlying system. This is both because we assume most readers are familiar with Prolog and because of the "good behaviour" of the Herbrand solver. However, we indicate which parts of the definitions are valid in any system and which other ones can only be used under certain assumptions.

The predicate `system_dependent_entailed` should succeed only if we can guarantee that `Prop` is implied by the store. This is probably the most system-dependent predicate of the four. Some constraint systems may have a complete entailment test, while others may not. In Herbrand this can be done by checking that `NProp`, which is a copy of `Prop`, succeeds and no new constraints (bindings) have been generated during its execution. This is checked by predicate `instance` which succeeds iff `Prop` is an instance of `NProp`:

```
system_dependent_entailed(Prop):-
        copy_term(Prop,NProp),
        call(NProp), !, instance(Prop,NProp).
```

Next we provide a definition for predicate `system_dependent_compatible`, which is valid for any constraint system which is "quick rejecting" (also called immediate), as is the case in Herbrand. In those systems, whenever we add a constraint which is inconsistent with the current store, this is immediately detected and a failure is issued. Note that this is not always the case since the system may delay the consistency check until further constraints added to the store allow performing the check in a simple way:

```
system_dependent_compatible(Prop):-
        \+(\+(call(Prop))).
```

Next we provide a definition of predicate `system_dependent_disentailed` which in contrast to the other three predicates is defined by two clauses. Each one provides a sufficient condition for disentailment. The first one corresponds to the case of incompatibility (whose definition appears later). Clearly, if `Prop` is incompatible with the current store it is also disentailed (but not the other way around). The second clause corresponds to the case in which execution of the property succeeds but it explicitly requires additional information in order to succeed. An example of this is the property `list(X)` which is an

instantiation type which requires `X` to be instantiated (at least) to a list skeleton, and with the store containing the information `X = [1|Y]`. In this case, `X` is compatible with a list (and thus the first clause would fail) but it is not actually a list. A call to `list([1|Y])` would succeed but would for example add the extra information that `Y=[]`:

```
system_dependent_disentailed(Prop):-
        system_dependent_incompatible(Prop),!.
system_dependent_disentailed(Prop):-
        copy_term(Prop,NProp),
        call(NProp), !, \+(instance(Prop,NProp)),
        instance(NProp,Prop).
```

Finally, for predicate `system_dependent_incompatible` we provide a definition which is valid in any system:

```
system_dependent_incompatible(Prop):-
        \+(call(Prop)).
```

2.6 A Sample Debugging Session with the Ciao System

In order to illustrate from the users point of view several of the points made in the previous sections we now present a sample debugging session with a concrete tool, CiaoPP, the Ciao system preprocessor [2.30, 2.8] which is currently part of the programming environment of Ciao, and which, as mentioned before, is an instance of the generic preprocessor presented in this chapter. CiaoPP uses as analysers both the LP and CLP versions of the PLAI abstract interpreter [2.38, 2.6, 2.27] and adaptations of Gallagher's regular type analysis [2.26], and works on a large number of abstract domains including moded types, definiteness, freeness, and grounding dependencies (as well as more complex properties, such as bounds on cost, determinacy, non-failure, etc., for Prolog programs).

Basic Static Debugging. The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 2.6. The result of regular type analysis for this program includes the following code:

```
:- true pred qsort(A,B)
         : ( term(A), term(B) )
        => ( list(A,t113), list(B,t118) ).

:- regtype prop t113/1.
t113(A) :- arithexpression(A).
t113([]).
t113([A|B]) :- arithexpression(A), list(B,t113).
t113(e).

:- regtype prop t118/1.
t118(x).
```

```
:- module(qsort, [qsort/2], [assertions]).

qsort([X|L],R) :-
        partition(L,L1,X,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[x|R1],R).
qsort([],[]).

partition([],_B,[],[]).
partition([e|R],C,[E|Left1],Right):-
        E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,   partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

Fig. 2.6. A tentative qsort program

where `arithexpression` is a library property which describes arithmetic expressions. Two new names (`t113` and `t118`) are given to types inferred, and their definition included, because no definition of such types were found visible to the module. In any case, the information inferred does not seem compatible with a correct definition of `qsort`, which clearly points to a bug in the program.

In order to debug the program, we add to it a declaration of its valid queries as follows:

```
:- entry qsort(A,B) : (list(A, num), var(B)).
```

Turning on compile-time error checking and selecting the `regtype` and `shfr` static analyses we obtain the following messages:

```
WARNING: Literal partition(L,L1,X,L2) at qsort/2/1/1
                 does not succeed!
ERROR: Predicate E>=C at partition/4/3/1 is not called as expected:
         Called:   num>=var
         Expected: arithexpression>=arithexpression
```

The first message warns that all calls to `partition` will fail, something normally not intended (e.g., in our case). The second message indicates a wrong call to a built-in predicate, which is an obvious error. This error has been detected by comparing the information obtained by the `shfr` inference system, which at the corresponding program point indicates that `E` is a free variable, with the assertion:

```
:- check calls A<B : (arithexpression(A), arithexpression(B)).
```

which is present in the default built-ins module, and which implies that the two arguments to `</2` should be bound to arithmetic expressions, and thus

ground. The message signals an *abstract* incorrectness symptom, indicating that the program does not satisfy the specification given (that of the built-in predicates, in this case). Checking the indicated call to `partition` and inspecting its arguments we detect that in the definition of `qsort`, `partition` is called with the second and third arguments in reversed order – the correct call is `partition(L,X,L1,L2)`.

After correcting this bug, we proceed to perform another round of compile-time checking, which produces the following message:

```
WARNING: Clause 'partition/4/2' is incompatible with its call type
         Head:      partition([e|R],C,[E|Left1],Right)
         Call Type: partition(list(num),num,var,var)
```

This time the error is in the second clause of `partition`. Checking this clause we see that in the first argument of the head there is an `e` which should be `E` instead. Compile-time checking of the program with this bug corrected does not produce any further warning or error messages.

Validation of User Assertions. In order to be more confident about our program, we add to it a partial specification of the program in the form of the following `check` assertions:

```
:- shfr prop ground/1.
:- regtype prop list/2.
:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- number(X).
sorted_num_list([X,Y|Z]):-
        number(X), number(Y), X<Y, sorted_num_list([Y|Z]).

:- calls qsort(A,B) : list(A, num).                          % A1
:- success qsort(A,B)  => (ground(B), sorted_num_list(B)). % A2
:- calls partition(A,B,C,D) : (ground(A), ground(B)).       % A3
:- success partition(A,B,C,D) => (list(C, num),ground(D)). % A4
:- calls append(A,B,C) : (list(A,num),list(B,num)).         % A5
```

where we also use a new property, `sorted_num_list`, defined in the module itself. We then ask CiaoPP to check them, by comparing them with the information inferred by analysis, which produces:

```
:- checked calls qsort(A,B) : list(A,num).                          %A1
:- check success qsort(A,B)  => sorted_num_list(B).                 %A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).        %A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D)). %A4
:- false calls append(A,B,C) : (list(A,num),list(B,num)).           %A5
```

Assertion `A5` has been detected to be false. This indicates a violation of the specification given, which is also flagged by CiaoPP as follows:

```
ERROR: (lns 22-23) false calls assertion:
   :- calls append(A,B,C) : list(A,num),list(B,num)
      Called append(list(^x),[^x|list(^x)],var)
```

The error is now in the call `append(R2,[x|R1],R)` in `qsort` (`x` instead of `X`). After correcting this bug and preprocessing the program again we get:

```
:- checked calls qsort(A,B) : list(A,num).                          %A1
:- check success qsort(A,B)  => sorted_num_list(B).                 %A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).        %A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D)). %A4
:- checked calls append(A,B,C) : (list(A,num),list(B,num)).         %A5
```

I.e., assertions `A1`, `A3`, `A4`, and `A5` have been validated but it was not possible to prove statically assertion `A2`, which has remained with `check` status, though it has been simplified. On the other hand the analyses used in our session (`regtypes` and `shfr`) do not provide enough information to prove that the output of `qsort` is a *sorted* list of numbers, since this is not a native property of the analyses being used. While this property could be captured by including a more refined domain (such as constrained types), it is interesting to see what happens with the analyses selected for the example.[18]

Dynamic Debugging with Run-time Checks. Assuming that we stay with the analyses selected previously, the following step in the development process is to compile the program obtained above with the "generate run-time checks" option. In the current implementation of CiaoPP we obtain the following code for predicate `qsort` where the new predicate name generated is `qsort_1`:

```
qsort(A,B) :-
        qsort_1(A,B),
        postc([ (qsort(C,D), true, sorted(D)) ], qsort(A,B)).

qsort_1([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R).
qsort_1([],[]).
```

where the code for `partition` and `append` remain the same as there is no other assertion left to check. If we now run the program with run-time checks in order to sort, say, the list `[1,2]`, the Ciao system generates the following error message:

```
?- qsort([1,2],L).
ERROR: for Goal qsort([1,2],[2,1])
Precondition: true  holds, but
Postcondition: sorted_num_list([2,1]) does not.

L = [2,1] ?
```

Clearly, there is a correctness problem with `qsort`, since `[2,1]` is not the result of sorting `[1,2]` in ascending order. This is a (now, run-time, or *concrete*)

[18] Note that this property, although not provable with the analyses selected, it is however disprovable.

incorrectness symptom, which can be used as the starting point of diagnosis. The result of such diagnosis should indicate that the call to `append` (where `R1` and `R2` have been swapped) is the cause of the error and that the right definition of predicate `qsort` is the following:

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).
```

Other CiaoPP Functionalities. In addition to the debugging-related functionality discussed before, CiaoPP includes a number of other capabilities related to the application of analysis results to program optimisation, such as program specialisation and program parallelisation. While most of these functionalities are in general outside our current scope, we will discuss a particular one, abstract (multiple) specialisation [2.41, 2.40]. As we will see later, this type of optimisation, performed as a source to source transformation of the program, and in which static analysis is instrumental, is indeed relevant in our context.

Program specialisation optimises programs for known values (substitutions) of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialisation can still be performed in such cases by means of abstract interpretation, specialisation then being with respect to abstract values, rather than concrete ones. Such abstract values represent of a (possibly infinite) set of concrete values. For example, consider the definition of the property `sorted_num_list/1`, and assume that regular type analysis has produced:

```
:- true pred sorted(A) : list(A,num) => list(A,num).
```

Abstract specialisation can use this information to optimise the code into:

```
sorted_num_list([]).
sorted_num_list([_]).
sorted_num_list([X,Y|Z]):- X<Y, sorted_num_list([Y|Z]).
```

which is clearly more efficient because no `number` tests are executed. The optimisation above is based on "abstractly executing" the `number` literals to the value `true`, the same notion used in reducing assertions during compile-time checking.

The importance of abstract specialisation in our context relies partly in that the availability of the abstract specialiser [2.41, 2.40] allows an alternative implementation of the whole framework (also using both compile-time and run-time checking of assertions) by first generating in a naive way a program which performs run-time checking of all assertions and then applying the abstract specialiser to this program. The resulting code would be similar to that obtained with the previous approach (first simplifying the assertions

in a specialised module and then generating code for those which cannot be statically proved): `checked` assertions will result in run-time tests that are optimised away, `false` assertions will result in run-time tests that are transformed to `error`, etc. However, we have opted for the first alternative because we have found that it is easier for the user to understand things in terms of simplified assertions rather than by looking at the run-time tests which remain in the transformed code.

2.7 Some Practical Hints on Debugging with Assertions

As mentioned before, one of the main features of the preprocessor we present is that assertions are optional and can state partial specifications. The fact that assertions are optional has important consequences on the ease of use and the practicality of the whole approach. An important drawback of many verification systems is the need for a relatively precise specification of the program. Writing such a specification is usually a tedious and not straightforward task. As a result, users in practice often get discouraged and may decide not to use systems which require quite detailed specifications. In contrast, in our framework assertions can be written "on demand", perhaps adding them only for those predicates, program points, and properties that the user wants to check in a given program. Clearly, as more (and more precise) assertions are added to a program, more bugs can potentially be detected automatically. Note that during the process of program development and debugging we will often turn our attention from some parts of the programs to others, and thus the set of assertions may change from one iteration to another.

The fact that assertions are optional obviously raises questions regarding issues such as, for which parts of the program should one write `check` assertions, what kinds of assertions should be used for a given objective, which kind of properties should be used in a given assertion, etc. Many of these questions are still open for research. Nevertheless, we can attempt to provide a few answers.

A point to note is that, from the point of view of their use in debugging, `calls` assertions are conceptually somewhat different from `success` and `comp` assertions. It is not of much use to introduce `success` and `comp` assertions during debugging for predicates which are known to be correct. Introducing `success` and `comp` assertions is in general most useful for *suspect* predicates. On the other hand, introducing `calls` assertions is a good idea even for correct predicates because the fact that a predicate is correct does not guarantee that it is called in the proper way in other parts of the program.

An important remark is that it is usually the case that different parts of the program are perceived by the user as having different levels of reliability [2.23]. For example, in order to detect a bug it is usually good practice to assume that library predicates are correct. For a tool to be successful, we believe that such different levels of reliability should somehow be reflected

during the validation/debugging session so that the programmer's attention can concentrate on a particular part of the code. Otherwise the debugging task becomes unrealistic for real programs. This can be achieved in our framework by adding assertions for those predicates that attention is focussed on and by "removing" assertions for others which are no longer under consideration. One very sensible way of doing this is by using modules. Dividing a program into modules allows performing compile-time checking by focusing on a single module, while not judging the code in other modules, of which we are only aware through a high-level description of the imported predicates (i.e., assertions for internal predicates of an imported module are effectively "turned off"). This is the approach used in CiaoPP.

Acknowledgements

The authors would like to thank Saumya Debray and Lee Naish for many interesting discussions on static analysis and debugging, Pawel Pietrzak for his adaptation of John Gallagher's type analysis for CLP(FD), Pedro López for greatly improving the interface to the above mentioned type analysis and making type information available not only at predicate level but also at each point in the program, and Daniel Cabeza for implementing the extended syntactic checking performed by the preprocessor. This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 "EDIPIA" and TIC97-1640-CE.

References

2.1 A. Aggoun and N. Beldiceanu. Overview of the chip compiler system. In *Proc. International Conference on Logic Programming*, pages 775–789. The MIT Press, 1991.

2.2 F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

2.3 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

2.4 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.

2.5 F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

2.6 F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

2.7 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
2.8 F. Bueno, P. López, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, November 1999.
2.9 F. Bueno, P. López, G. Puebla, M. Hermenegildo, and P. Pietrzak. The CHIP Assertion Preprocessor. Technical Report CLIP1/99.1, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, March 1999. Also as deliverable of the ESPRIT project DISCIPL.
2.10 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.
2.11 D. Cabeza and M. Hermenegildo. A Modular, Standalone Compiler for Ciao Prolog and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.
2.12 D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.
2.13 B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.
2.14 M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
2.15 M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.
2.16 The COSYTEC Team. *CHIP System Documentation*, April 1996.
2.17 P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
2.18 P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.
2.19 S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. The MIT Press, Cambridge, MA.
2.20 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.
2.21 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyǹski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. The MIT Press, 1989.
2.22 M. Ducassé. OPIUM - an advanced debugging system. In M. J. Comyn, G.; Fuchs, N.E.; Ratcliffe, editor, *Proceedings of the Second International Logic*

Programming Summer School on Logic Programming in Action (LPSS'92), volume 636 of *LNAI*, pages 303–312, Zürich, Switzerland, September 1992. Springer-Verlag.

2.23 M. Ducassé. A pragmatic survey of automated debugging. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, May 1993.

2.24 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

2.25 J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

2.26 J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

2.27 M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

2.28 M. Hermenegildo. A System for Automatically Generating Documentation for (C)LP Programs. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science, 2000.

2.29 M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

2.30 M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. The MIT Press.

2.31 M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. The MIT Press. (abstract of invited talk).

2.32 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

2.33 M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

2.34 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

2.35 N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall, New York, 1993.

2.36 A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. The MIT Press, 1996.

2.37 A. Kelly, K. Marriott, H. Søndergaard, and P.J. Stuckey. A generic object oriented incremental analyser for constraint logic programs. In *Proceedings of the 20th Australasian Computer Science Conference*, pages 92–101, 1997.
2.38 K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
2.39 G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS. Springer-Verlag, 2000. To appear.
2.40 G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.
2.41 G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
2.42 G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
2.43 E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. The MIT Press, 1982.
2.44 E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

3. Assertions with Constraints for CLP Debugging

Claude Laï

Société PrologIA, Parc Technologique de Luminy,
Case 919, 163 Avenue de Luminy, 13288 Marseille Cedex 9, France
email: lai@lim.univ-mrs.fr

This chapter describes the use of partial correctness assertions for CLP debugging. These assertions are formed by a set of preconditions and a set of postconditions associated with each predicate used in the program. We present a system with a rich set of primitives (based on constraints and meta-properties) to express the program properties. These primitives are such that they can often be proven efficiently with a local analysis at compile-time. In case of failure of the proof procedure, the program is completed with run-time checks to insure a correct execution.

3.1 Introduction

It is considered there are two important aspects to help the debugging of CP applications: first, the choice of a programming methodology, and second, the selection of a validation procedure. Validation is the verification of the correspondence between the program and the programmer's intention. It can be modelled by the notions of partial correctness of programs and completeness of solutions. The main problem is to locate bugs when symptoms of incorrectness or incompleteness have been observed. Since CP is a declarative paradigm, it is particularly well suited to apply declarative analysis methods such as assertion based methods used in validation [3.5], and declarative debugging techniques used to locate bugs [3.1, 3.8, 3.7].

As a matter of fact, the introduction of assertions in programs addresses both of these aspects. Assertions are partial specifications which are provided by the programmer. They can be attached to the program predicates and they represent expected meaning of the predicates. Assertions are associated to the notion of partial correctness . A special kind of assertions, concerning the execution of a program with the Prolog computation rule, has been discussed in [3.6]. The described properties are sometimes called dynamic or run-time properties. A special kind of run-time assertions, where the verification conditions becomes simpler, is known under the name of directional types [3.3].

The recent work of [3.2] describes a refinement of the verification conditions for directional types, and [3.9] shows that these types are applicable for the debugging of CP programs, where the assertions are considered as constraints on an extended constraint structure, their proof being performed

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 109–120, 2000.

using abstract interpretation techniques. This chapter describes a continuation of this work based on the same ideas. The practical aspect of this work is obviously to give a powerful debugging tool to the user. This tool allows to introduce a good specification of programs and performs automatic verifications of these programs.

In this chapter, we will be referring to Prolog with constraints as the language used to write the programs we want to prove. Prolog IV [1] is the support used for the examples of constraints.

This chapter is organised as follows: first, the form of the assertions is defined; then a verification of the program at compile time is described; finally a transformation of the program is presented which allows a fair execution of rules which assertions could not be proved.

3.2 Form of Assertions

3.2.1 Syntax and Meaning of Assertions

Assertions are expressed with a new directive handled by a compiler preprocessor. For a given predicate, an assertion is formed by a set of preconditions and a set of postconditions. It is described in the following syntax:
$:- pred(X_1, X_2, ..., X_n) \ : \ (pre_1, pre_2, ..., pre_p) \ => \ (post_1, post_2, ..., post_q).$
where:

- $pred$ is the name of the predicate;
- $X_1, X_2, ..., X_n$ are distinct variables representing the arguments of the predicate;
- $pre_1, pre_2, ..., pre_p$ represent the set of preconditions;
- $post_1, post_2, ..., post_q$ represent the set of postconditions.

The interpretation of such an assertion is:

- the set of preconditions is interpreted as a conjunction and defines the needed properties of an admissible model for the given definition of the predicate $pred$;
- the set of postconditions is interpreted as a conjunction of the properties of the set of solutions satisfying the predicate $pred$ in that model.

In an operational view, assertions have a meaning very similar to the procedure contracts in the Eiffel language: the caller must guarantee the truth of the preconditions before the predicate call, and the predicate definition must guarantee that the postconditions will hold after execution.

[1] Prolog IV is an operational product developed and marketed by PrologIA, Parc scientifique de Luminy, case 919, 13288 Marseille Cedex 9, France. URL: http://prologianet.univ-mrs.fr

3.2.2 The Basic Constructs

We have now to define the expected form of the conditions in the assertions. In our case, it is possible to mix two different kinds of conditions: atomic constraints of Prolog IV solvers which admit an opposite constraint, and more general "meta" conditions which we will handle with abstract domains. Prolog IV has two main solvers for operations on numbers: a linear solver which is complete, and an interval solver (handling a very rich set of linear and non linear operations) which is not. However, there is, for the interval solver, an extension of the usual structure, for which this solver is complete. It is such that any solution in the actual structure is also a solution in the extended structure and, if there is no solution in the extended structure, there is also no solution in the real structure. Therefore, a classical proof by inconsistency with the negation of a property is sound with this solver too: if there is no solution in the extended structure it is guaranteed there is none in the real one.

Atomic Constraints.

- Domain constraints:
  - $int(X)$ constrains X to only have integer solutions;
  - $nint(X)$ constrains X not to have integer solutions;
  - $real(X)$ constrains X to only have number solutions;
  - $nreal(X)$ constrains X not to have number solutions;
  - $list(L)$ constrains L to only have list solutions;
  - $nlist(L)$ constrains L not to have list solutions;
- Interval constraints:
  - $cc(X, B_1, B_2)$ which constrains X solutions to belong to the interval $[B_1, B_2]$;
  - $outcc(X, B_1, B_2)$ which constrains X solutions to be out of the interval $[B_1, B_2]$;
  - $co(X, B_1, B_2)$ which constrains X solutions to belong to the interval $[B_1, B_2)$;
  - $outco(X, B_1, B_2)$ which constrains X solutions to be out of the interval $[B_1, B_2)$;
  - $Z = K\ .*.\ Y$, with K a constant number. This relation is handled by the interval solver. It establishes a relationship between two numbers X and Y and their product Z;
  - $Z = X\ .+.\ Y$. This relation is handled by the interval solver. It establishes a relationship between two numbers X and Y and their sum Z;
  - $X\ le\ Y$ ("less or equal" handled by the interval solver);
  - $X\ gt\ Y$ ("greater than" handled by the interval solver);
  - other similar comparison operators ...
- Linear constraints:
  - $Z = K * Y$, with K a constant number;

  - $Z = X + Y$;
  - X *lelin* Y ("less or equal" handled by the linear solver);
  - X *gtlin* Y ("greater than" handled by the linear solver);
  - other similar comparison operators ...
- General constraints like $dif(X, Y)$ which constrains X and Y to be distinct (disequation).

Meta-Properties. The second category of conditions contains "meta" constraints or properties.

- $ground(X)$ which is true if and only if the set of elements of the domain, which can be assigned to X and which satisfies the constraints, is a singleton.
- $listnum(L)$ which constrains L to be a list of numbers.
- $listnum(L, A, B)$ which constrains L solutions to be a list of numbers in the interval $[A, B]$.

3.3 Correctness Proofs

The goal is to prove that a given program is correct, according to the assumed assertions for the predicates in this program.

3.3.1 Implementation

The method used here, is to verify each rule independently, considering all the assertions of the program. That means a local analysis is used rather than a global one. The reasons are that it is simpler and more efficient, it matches an incremental compilation model very well. If for each rule, each call in the body fulfils the preconditions, and if the postconditions are implied, the program is correct. Therefore, for each rule of the program, the following is performed:

1. The current state of constraints is initialised to empty;
2. the preconditions of the corresponding predicate are added to the current state;
3. for each call which occurs in the body, if it is a constraint, it is added to the current state, otherwise:
   - the preconditions of the called predicate are checked (that is to say they are implied by the current state);
   - the postconditions of the called predicate are added to the current state (that is to say we assume the predicate definition is correct).
4. the postconditions of the rule are checked (to be implied).

Checking and adding conditions are performed by a meta-interpreter of assertions.

Depending on the class of the condition (atomic constraint or meta-property) a different procedure is used. For atomic constraints, a proof by inconsistency is performed, that is to say the opposite constraint is temporarily added to the current state. If the new state is not solvable, this means that the original condition is implied and the check is satisfied. Otherwise, a solution exists in the extended structure which means that the original condition is not implied by the current state and the proof fails.

For meta-properties the proof is done by an abstract interpreter in a classical way. In fact, both computations are done in a combined domain abstraction.

3.3.2 Example of a Verification

We do not give here a complete formalisation of the abstractions used. We just present a sketch of these abstractions. At present, each term is abstracted by two constraint systems (following the ideas described in [3.4, 3.9]). The first abstract domain is used to study a meta-property (groundness). The second one is used to abstract the actual domain and the atomic constraints.

As an example a variable X will be abstracted by the list $[X_g, X_x]$ where:

- X_g is a boolean variable representing the groundness property;
- X_x is a representation of the atomic constraint state.

Let us consider now this program:

```
:- a(N,P) : (ground(N), cc(N,1,10)) => (ground(P), cc(P,10,100)).
:- b(N,P) : (true) => (P = 10 * N).
a(N,P) :-
    b(N,P).

...
```

In this example, we want to prove the assertion for the rule $a/2$. The prover abstracts the variable N by the structure $[N_g, N_x]$ and the variable P by the structure $[P_g, P_x]$. According to the method described above, the following is performed:

1. The current state of constraints is initialised to empty;
2. The preconditions of $a/2$ are added to the current state. Thus, the abstraction of the condition $ground(N)$ has the effect to affect the boolean variable N_g to 1, and the abstraction of the condition $cc(N, 1, 10)$ has the effect to constrain the variable N_x to belong to the interval $[1, 10]$;
3. Nothing is checked for the call to the predicate of the body ($b/2$), because there is no precondition for it. Therefore, the postcondition $P = 10 * N$ is added. The abstraction of the operator '*' is defined as follows:

$$\begin{array}{l} timesK([Z_g, Z_x], [1, K], [Y_g, Y_x]) \text{ :-} \\ \quad Y_g = Z_g, \\ \quad Z_x = K * Y_x. \end{array}$$

So, the constraint $P_x = 10 * N_x$ is added to the current state.

4. The postconditions of $a/2$ are checked.
   - Given the fact that $N_g = 1$ and that the constant 10 is ground, the equality of the rule $timesK/3$ involves that P is ground ($P_g = 1$). The opposite condition of $ground/1$ consists in adding the abstraction $P_g = 0$. Obviously, this operation will fail.
   - To check the condition $cc(P, 10, 100)$, the prover will try to add the opposite condition $outcc(P_x, 10, 100)$ to the current state. The result will be a failure of the interval solver.

So, we have checked the set of postconditions for $a/2$. Thus, the assertion of this rule is proved.

3.3.3 Incompleteness Introduced by the Solvers

For the conditions which are atomic constraints, the completeness in the (real) intended domain is dependent on the solver involved. Let us consider the two Prolog IV solvers: the linear solver, which is complete, and the interval solver, which is incomplete. Depending on the solver used, a proof will succeed or not. As an example:

```
:- a(X,Y) : (X gelin Y) => (X gtlin Y).
a(X,Y) :- dif(X,Y).
```

In this example, the assertion of the rule $a/2$ can be proved. As a matter of fact, the constraints involved by the set of preconditions are first added to the current set of constraints. Thus, the constraint $X\ gelin\ Y$ is added. When the body of the rule is handled, the constraint $dif(X, Y)$ is added. Then, to verify the postcondition, the goal is to check if the postcondition is implied by the current set of constraints. To do that, the opposite constraint (here the constraint $X\ lelin\ Y$) is added to the current set, and a failure is expected. Given the completeness of the linear solver, this failure is obtained. So, the implication of the postcondition by the current set is proved.

Now, let us consider the example:

```
:- b(X,Y) : (X ge Y) => (X gt Y).
b(X,Y) :- dif(X,Y).
```

For the rule $b/2$, the behaviour is not the same. When the opposite constraint $X\ le\ Y$ is added to the current set (which already contains the constraint $dif(X, Y)$ added by the treatment of the body), the expected failure does not occur. So, the assertion is not proved.

A similar behaviour can appear when both solvers are requested in one assertion. Example:

```
:- d(X,Y) : (X gtlin 3) => (Y gt 5).
d(X,Y) :- Y = X + 2.
```

If one wants to reduce the effects of the incompleteness of the interval solver, it is advisable in assertions:

- to limit non linear constraints to have a unique variable (and other constants) among their arguments;
- not to mix constraints handled by different solvers.

More generally, if we want to guarantee soundness of our system, we have to use only constraints handled by complete solvers. In a further work, we will try to specify exactly what conditions have to be filled by the constraints (in general and in our particular case with these two solvers) to avoid some basic failures of the proofs.

3.3.4 Full Example

Let us consider the famous program $SEND + MORE = MONEY$. This program can be entirely proved if we introduce assertions taking into account the considerations listed above. That gives the following source code:

```
:- solution(Send,More,Money) : (true) =>
       (ground(Send), cc(Send, 1000,9999),
        ground(More), cc(More, 1000,9999),
        ground(Money), cc(Money, 10000,99999)).
:- allintegers(L) : (true) => (ground(L), listnum(L)).
:- allDifferentDigits(L) : (listnum(L)) => (listnum(L,0,9)).
:- outsideOf(X,L) : (listnum(L) , real(X)) => (true).
:- difrat(X,Y) : (real(X), real(Y)) => (true).
:- enum(X) : (true) => (ground(X), real(X)).

allintegers([]) .
allintegers([E|R]) :- enum(E), allintegers(R).

allDifferentDigits([]) .
allDifferentDigits([X|S]) :-
        0 le X, X le 9,
        outsideOf(X, S),
        allDifferentDigits(S).

outsideOf(X, []) .
outsideOf(X, [Y|S]) :- difrat(X,Y), outsideOf(X, S).
```

```
solution(Send, More, Money) :-
        S ge 1 ,
        M ge 1 ,
        Send .+. More = Money ,
        Send = 1000 .*. S .+. 100 .*. E .+. 10 .*. N .+. D ,
        More = 1000 .*. M .+. 100 .*. O .+. 10 .*. R .+. E ,
        Money = 10000 .*. M .+. 1000 .*. O .+. 100 .*. N .+.
                10 .*. E .+. Y ,
        allDifferentDigits([M,S,O,E,N,R,D,Y]),
        allintegers([M,S,O,E,N,R,D,Y]) .
```

3.3.5 Samples of Compilations

This program is correct:

```
:- fact(N,P) : (bound(N),real(N)) => (bound(P),real(P)).

fact(0,1) :- !.
fact(N,N*P) :-
    fact(N-1,P).
```

This program is not correct, according to the assertion:

```
:- badfact(N,P) : (bound(N),real(N)) => (bound(P),real(P)).
badfact(0,W) :- !. /* ¡– The error is here */
badfact(N,N*P) :-
    !,
    badfact(N-1,P).
WARNING : Postcondition verification failed for badfact/2
        (Assertion No 1: bound(_3781)
        Rule checked: badfact/2 Rule nb: 1)
WARNING : Postcondition verification failed for badfact/2
        (Assertion No 2: real(_3998)
        Rule checked: badfact/2 Rule nb: 1)
```

These compilations succeed without any warning:

```
:- sumlist(L,S) : (listnum(L),ground(L)) =>
    (ground(S),real(S)).
sumlist([],0).
sumlist([X|L],S1) :-
    S1 = S + X,
    sumlist(L,S).
```

```
:- sl(L,S) : (listnum(L,0,100000), ground(L)) =>
     (ground(S), Sge0).
sl([],0).
sl([X|L],X.+.S) :-
     sl(L,S).
```

```
:- slOk(L,S) : (ground(L)) => (ground(S)).
slOk([],0).
slOk([X|L],S1) :-
     S1 = S + X,
     slOk(L,S).
```

This compilation raises warnings for groundness (Postcondition):

```
:- slwrong(L,S) : (list(L)) => (ground(S),real(S)).
slwrong([],0).
slwrong([X|L],S1) :-
     S1 = S + X,
     slwrong(L,S).
WARNING: Postcondition verification failed for slwrong/2
          (Assertion No 1: ground(_7618)
          Rule checked: slwrong/2 Rule nb: 2)
```

In this compilation, the proof fails because it needs an undefined assertion:

```
:- slwrong(L,S) : (listnum(L),ground(L)) =>
     (ground(S),real(S)).
slwrong([],0).
slwrong([X|L],S1) :-
     S1 = S + X,
     slmiss(L,S).
WARNING: assertion is missing for slmiss/2
WARNING: Postcondition verification failed for slwrong/2
          (Assertion No 1: ground(_7163)
          Rule checked: slwrong/2 Rule nb: 2)
```

This compilation raises warnings for groundness (Precondition):

```
:- slBad(L,S) : (ground(S)) => (ground(L)).
slBad([],0).
slBad([X|L],S1) :-
     S1 = S + X,
     slBad(L,S).
WARNING: Precondition verification failed for slBad/2
          (Assertion No 1: ground(_6952)
          Rule checked: slBad/2 Rule nb: 2)
```

This program has an inconsistent assertion:

$:-\ slerror(L,S) : (list(L), real(L)) => (ground(S), real(S)).$
$slerror([],0).$
$slerror([X|L],S1)\ :-$
$\quad S1 = S + X,$
$\quad slerror(L,S).$
$slerror/2 : real(\_3237)$ Rule checked: $slerror/2$ Rule nb: 1
error : ['A clause or assertion is inconsistent']

3.4 Run-Time Checking

When verifying a program, if a precondition cannot be proved for a given goal in the body of a rule, the goal is flagged as unchecked for this precondition. Then, the proof continues normally as if the proof of all the preconditions had succeeded. At compile-time, the code needed to verify the unchecked preconditions is inserted before the goal. The principle is the same for the postconditions. Most of the time, a run-time check is easier to perform than a static general proof because the constraints accumulated by the execution of the whole program specify arguments. Example:

```
:- sumlist(L,S) : (list(L)) => (ground(S), real(S)).

sumlist([],0).
sumlist([X | L],S1) :-
        S1 = S + X ,
        sumlist(L,S).
```

In this example which computes the sum of a list, no groundness is assumed by the preconditions, so for the second rule the groundness cannot be proved in the postconditions. The rules actually compiled are:

```
sumlist([],0).
sumlist([X | L],S1) :-
        S1 = S + X ,
        sumlist(L,S),
        run_time_check_post(sumlist([X | L],S1)).
```

Samples of executions:

$sumlist([1,2,3],L)$.

$L = 6$.

$sumlist([Q,2,3],L)$.
WARNING: Postcondition verification failed for $sumlist/2$ Rule Nb: 2
Assertion No 1: $ground(\_967)$
$L = real$,
$Q = real$.

For the same rule, if we replace its unique assertion by:

```
:- sumlist(L,S) : (ground(S)) => (ground(L)).
```

the groundness in the precondition cannot be proved for the goal $sumlist/2$ called in the body of the second rule, so the rules actually compiled will be:

```
sumlist([],0).
sumlist([X | L],S1) :-
        S1 = S + X ,
        run_time_check_pre(sumlist(L,S)),
        sumlist(L,S).
```

where $run\_time\_check\_post/1$ and $run\_time\_check\_pre/1$ are procedures which retrieve the postconditions (or preconditions) of its predicate argument and check them.

Samples of executions:

$sumlist([1,2,3],K)$.
WARNING: precondition verification failed for call of $sumlist/2$
in rule $sumlist/2$ Rule Nb: 2 (Assertion No 1: $ground(\_950)$)
WARNING: precondition verification failed for call of $sumlist/2$
in rule $sumlist/2$ Rule Nb: 2 (Assertion No 1: $ground(\_1002)$)
WARNING: precondition verification failed for call of $sumlist/2$
in rule $sumlist/2$ Rule Nb: 2 (Assertion No 1: $ground(\_1054)$)
$K = 6$.
$sumlist([1,2,3],6)$.
true.

3.5 Conclusion

We have presented an extension to constraint languages consisting in introducing assertions. This extension allows an automatic verificationof programs,

combining compile-time analysis and run-time verifications. The assertions can be expressed with a rich set of primitives, including constraints, and are very helpful for program debugging. The benefit is also to introduce a much better specification and readibility of programs. Such an extension has been introduced in the Prolog IV environment and is a very valuable development tool. It has been validated on complex real-life applications solving large scale problems.

References

3.1 G. Ferrand, A. Tessier. Positive and Negative Diagnosis for Constraint Logic Programs in terms of proof skeletons, AADEBUG'97, 1997.
3.2 J. Boye, J. Małuszyǹski. Two aspects of Directional Types In Proc. Twelfth Int. Conf. on Logic Programming, pp. 747-763, The MIT Press, 1995.
3.3 F. Bronsart, T.K. Lakshman, U. Reddy. A framework of directionality for proving termination of logic programs. In Proc. of JICSLP'92, pp. 321-335. The MIT Press, 1992.
3.4 P. Codognet, G. Filé. Computations, Abstractions and Constraints in Logic Programs, International Conference on Computer Language, Oakland, 1992.
3.5 P. Deransart, J. Małuszyǹski. A grammatical view of logic programming. The MIT Press, 1993.
3.6 Drabent, W., J. Małuszyǹski. Induction assertion method for logic programs. Theorical Computer Science 59, pp. 133-155, 1998.
3.7 G. Ferrand. Error Diagnosis in Logic Programming. Journal of Logic Programming 4, 177-198, 1987.
3.8 F. Le Berre, A. Tessier. Declarative Incorrectness Diagnosis of Constraint Logic Programs. Rapport de Recherche LIFO 95-08 Université d'Orléans, 1995.
3.9 E. Vétillard. Utilisation de déclarations en Programmation Logique avec Contraintes, PhD thesis, U. of Aix-Marseilles II, 1994.

4. Locating Type Errors in Untyped CLP Programs

Włodzimierz Drabent[1,2], Jan Małuszyński[1], and Paweł Pietrzak[1*]

[1] Linköpings universitet, Department of Computer and Information Science
S – 581 83 Linköping, Sweden
email: {wdr|jmz|pawpi}@ida.liu.se
[2] Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21,
Pl – 01-237 Warszawa, Poland

This chapter presents a *static diagnosis* tool that locates type errors in untyped CLP programs without executing them. The existing prototype is specialised for the programming language CHIP [4.10], but the idea applies to any CLP language. The tool works with approximated specifications which describe types of procedure calls and successes. The specifications are expressed as a certain kind of term grammars. The tool automatically locates at compile time all the errors (with respect to a given specification) in a program. The located erroneous program fragments are (prefixes of) clauses. The tool aids the user in constructing specifications incrementally; often a fragment of the specification is already sufficient to locate an error. The presentation is informal. The focus is on the motivation of this work and on the functionality of the tool. Some related formal aspects are discussed in [4.15, 4.29]. The prototype tool is available from http://www.ida.liu.se/~pawpi/Diagnoser/diagnoser.html.

4.1 Introduction

In a traditional setting, the process of locating an error starts with a *symptom* observed when running the program on test input data. A symptom is a discrepancy between some user expectations and the behaviour of the program at hand, e.g. a wrong computed answer or a procedure call with arguments which are outside the intended domain of application. After a symptom has been obtained, one wants to locate the *error* that is a minimal fragment of the program causing the symptom.

A rather ad hoc approach to locating an error is tracing the execution which shows a symptom. In the case of declarative languages, tracing is particularly difficult because the execution steps are rather complex and not reflected explicitly in the program. A more systematic technique for locating errors is *declarative* or *algorithmic* debugging proposed in [4.31] for logic programs (see also [4.18, 4.25]). Declarative diagnosis of CLP programs is studied in Chapter 5 of this volume.

* The authors acknowledge the contribution of Marco Comini who was largely involved in the development of an early version of the diagnosis tool [4.6, 4.7].

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 121–150, 2000.

In contrast to the above mentioned approaches, we propose to locate errors in a CLP program without searching for symptoms, that is without executing the program. The idea can be linked to methods for proving partial correctness of a program with respect to a specification, such as [4.5, 4.12, 4.13, 4.14, 4.1]. Our tool tries to construct a proof that the program is correct w.r.t. the specification. If the proof is obtained then every execution will be free of symptoms violating the specification. Conversely, if a symptom violating the specification can be observed, a proof does not exist. In an incorrect program our tool finds all the fragments that are responsible for non existence of the proof. To use the tool the user need not be familiar with the underlying program verification techniques.

We wanted to make the diagnoser as easy to use as possible. To achieve this it was necessary:

– to choose a simple specification language easy to understand by the user,
– to minimise the specification effort necessary to locate an error,
– to allow diagnosis of separate fragments of programs,
– to provide a convenient user interface.

We focus on the question whether incorrect procedure calls and wrong answers may appear in some computations of a given program. The specification provided by the user describes a superset of the expected calls and a superset of the expected answers in computations of the program. The program may or may not satisfy these expectations. The above mentioned sets are described in terms of parametric types such as lists, trees, etc. The language of types is easy to understand and allows efficient checking of the verification conditions. On the other hand, this choice restricts the considered errors to type errors.

We deal with untyped CLP languages and, in contrast to a common practice in typed languages, we do not require types of program constructs to be specified a priori.

Figure 4.1 presents a general overview of the tool. The two main components are the *type analyser* and the *diagnoser*.

The analyser is used to infer types of the predicates in a given program. The role of the diagnoser is to locate the errors. Diagnosis may be requested if the inferred type of a predicate is different from that expected by the user. The diagnoser responds with a list of types on which the diagnosed type depends. In order to locate the error, some of the types in the list (in the worst case all of them) have to be specified by the user. The list of types is sorted in a way that aims at reducing the amount of interaction with the user.

The specification requests are represented by the question mark in Fig. 1. An error locating message is generated by the diagnoser as soon as a sufficient subset of the required types is specified. Providing further specification may result in locating more errors.

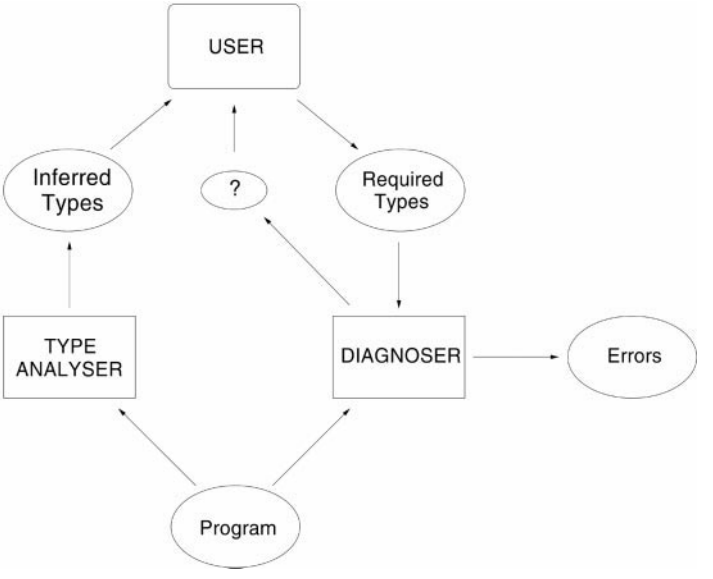


Fig. 4.1. Overview of the tool

The diagnoser finds incorrect clauses. Even more, it locates an error down to a clause prefix. It is able to locate *all* the errors in the program. In other words, all the reasons that the program behaves incorrectly (with respect to the specification) are within the located clause prefixes. Thus, if no errors are found then the program is correct. An obvious limitation is a restricted class of specifications.

The actual diagnosis is performed without referring to any symptoms. It refers neither to program execution nor to the results of program analysis. So the diagnoser can work without the analyser. The role of the analysis is auxiliary. It helps to discover that the program is possibly incorrect and to suggest a starting point of the diagnosis. The results of the analysis can be used as a draft for the specification; this simplifies the task of constructing the specification by the user.

The rest of this paper is organised as follows. Section 4.2 discusses the concept of partial correctness and the language of types as a basis for formulation of the diagnosis problem. The use of the tool and its functionality is illustrated on an example in Section 4.3. Section 4.4 explains informally the underlying principles of static diagnosis, Section 4.5 describes the treatment of delays in our approach and Section 4.6 discusses some implementation issues. Section 4.7 enumerates some limitations of our approach. Related work is surveyed in Section 4.8. Section 4.9 presents conclusions and future work.

4.2 The Specification Language

Intuitively, a program is correct if for any input data it behaves as expected by the user. For automatic support of (in)correctness analysis we need a language for describing both the program behaviour and user expectations. Computations of CLP programs are quite complex. Therefore we focus on selected aspects of computations, namely on calls and successes of program predicates in all computations. This section introduces a simple language for describing the form of predicate calls and successes. The actual behaviour of a program can be approximated by an automatically generated description in this language. The language will also be used for approximate specifications which describe user's expectations. Such specifications are used by our tool for automatic error location.

Notice that the form of procedure calls and successes is among the properties that can be described by assertions discussed in Chapter 1. The properties we deal with can be considered a special case of `calls` and `success` assertions, tagged with `true` or with `check` depending on whether the properties are obtained from program analysis or are a part of a specification. The actual ways of describing sets of constrained atoms, to which assertions refer, are however of secondary importance in Chapter 1. Here we introduce a formalism for describing a particular, suitable for our purposes, class of such sets.

Before introducing our specification language in Section 4.2.2, we explain how the behaviour of a CLP program is characterised in terms of predicate calls and successes. This includes discussing some basics of the chosen CLP semantics and presenting a formal definition of a procedure call and success. Some readers may prefer to skip the latter.

We assume a fixed CLP language (object language) over a fixed constraint domain $\mathcal{D}$. In the next subsection $\mathcal{D}$ is arbitrary, the rest of the paper deals with CLP over finite domains. To simplify the presentation, we first present our approach applied to programs without delays (i.e. executed under the Prolog selection rule)[1]. Treatment of delays is discussed in Section 4.5.

4.2.1 Calls and Successes of a CLP Program

In this section we present the semantics of CLP used in this work and introduce the notion of an approximate specification.

The errors we want to locate demonstrate themselves as wrong computed answers or as wrong arguments of predicate calls, where "wrong" refers to user expectations or to a priori given requirements concerning the use of built-ins. These aspects of computations can be captured by associating with

[1] We consider constraints as never delayed, assuming that each constraint is passed to the constraint solver as soon as it is selected by the Prolog selection rule. The internals of the solver are outside of the scope of our semantics.

each predicate two sets: one of them describing all calls and the other all successes in the considered class of computations. This section discusses this idea in more detail. For a more formal presentation see [4.29].

We consider CLP programs executed with the Prolog selection rule (LD-resolution) and using syntactic unification in the resolution steps. In CLP with syntactic unification, function symbols occurring outside of constraints are treated as constructors. So, for instance in CLP over integers, the goal $p(4)$ fails with the program $\{p(2+2)\leftarrow\}$ (but the goal $p(X+Y)$ succeeds). Terms 4 and 2+2 are treated as not unifiable despite having the same numerical value. Also, a constraint may distinguish such terms. For example in many constraints of CHIP, an argument may be a natural number (or a "domain variable") but not an arithmetical expression. Resolution based on syntactic unification is used in many CLP implementations, for instance in CHIP and in SICStus [4.30].

Computations of a CLP program involve constraints. Therefore, predicate calls and successes take the form of *constrained atoms*. A constrained expression (atom, term, etc) is a pair of the form $c \,[]\, E$ where c is a constraint and E is an expression such that each free variable of c occurs in E. For example, $X :: 1..4 \,[]\, p(X, Y)$ is a constrained atom in CHIP notation[2]. For a c not satisfying the latter condition, $c \,[]\, E$ will be an abbreviation for $(\exists_{\ldots} c) \,[]\, E$ where the quantification is over all variables not occurring in E. A constrained expression $true \,[]\, t$ may be represented as t.

We are interested in *calls* and *successes* of program predicates in computations of the program. Both calls and successes are constrained atoms. A precise definition is given below taking a natural generalisation of LD-derivation as a model of computation.

An *LD-derivation* is a sequence $G_0, C_1, \theta_1, G_1, \ldots$ of goals, input clauses and mgu's (similarly to [4.26]). A goal is of the form $c \,[]\, A_1, \ldots, A_n$, where c is a constraint and $A_1, \ldots, A_n$ are atomic formulae (including atomic constraints). For a goal $G_{i-1} = c \,[]\, A_1, \ldots, A_n$, where A_1 is not a constraint, and a clause $C_i = H \leftarrow B_1, \ldots, B_m$, the next goal in the derivation is $G_i = (c \,[]\, B_1, \ldots, B_m, A_2, \ldots, A_n)\theta_i$ provided that θ_i is an mgu of A_1 and H, $c\theta$ is satisfiable and G_{i-1} and C_i do not have common variables. If A_1 is a constraint then $G_i = c, A_1 \,[]\, A_2, \ldots, A_n$ ($\theta_i = \epsilon$ and C_i is empty) provided that c, A_1 is satisfiable.

For a goal G_{i-1} as above we say that $c \,[]\, A_1$ is a *call* (of the derivation). The call succeeds in the first goal of the form $G_k = c' \,[](A_2, \ldots, A_n)\rho$ (where $k \geqslant i$) of the derivation. So $\rho = \theta_i \cdots \theta_k$. The *success* corresponding (in the derivation) to the call above is $c' \,[]\, A_1\rho$. For example, $X :: 1..4 \,[]\, p(X, Y)$ and $X :: [1, 2, 4] \,[]\, p(X, 7)$ is a possible pair of a call and a success for p defined by $p(X, 7) \leftarrow X \neq 3$.

[2] $X :: 1..4$ and $X :: [1, 2, 3, 4]$ are two alternative ways for describing $X \in \{1, 2, 3, 4\}$ in CHIP notation.

Notice that in this terminology constraints succeed immediately. If A is a constraint then the success of call $c \,[]\, A$ is $c, A \,[]\, A$, provided c, A is satisfiable. So we do not treat constraints as delayed; we abstract from internal actions of the constraint solver.

The *call-success semantics* of a program P, for a set of initial goals $\mathcal{G}$, is a pair $CS(P, \mathcal{G}) = (C, S)$ of sets of constrained atoms: the set of calls and the set of successes that occur in the LD-derivations starting from goals in $\mathcal{G}$. We assume without loss of generality that the initial goals are atomic.

So the call-success semantics describes precisely the calls and the successes in the considered class of computations of a given program. The question is whether this set includes "wrong" elements, unexpected by the user. To require a precise description of user expectations is usually not realistic. On the other hand, it may not be difficult to provide an approximate description $Spec = (C', S')$ where C' and S' are sets of constrained atoms such that every expected call is in C' and every expected success is in S'. We say that P with initial goals $\mathcal{G}$ is *partially correct* w.r.t. *Spec* iff $C \subseteq C'$ and $S \subseteq S'$. We will usually omit the word "partially".

Our tool uses a fixed specification language for writing descriptions of calls and successes. The precision of specification, and the errors which can be discovered are thus a priori restricted by the language. On the other hand, the simplicity of the language allows efficient automatic location of errors if the program is not correct w.r.t. a given specification. Moreover, the intelligibility of the language helps the user in taking right decisions during an interaction with the diagnoser.

4.2.2 Describing Sets of Constrained Atoms

The program errors considered in this work concern discrepancies between expected and actual calls and successes, in other words, between the intended and the actual call-success semantics of the program. Thus, in order to formulate a program specification (or to present results of the static analysis to the user) we need a language for describing sets of constrained atoms and constrained terms. This section presents the specification language used in our tool. In this presentation we do not distinguish between function symbols and predicate symbols (and between terms and atoms).

For the purposes of program analysis and diagnosis, we need to compute certain operations on sets: set intersection and union (possibly approximated), inclusion and emptiness, and operations of construction and deconstruction which are explained in Section 4.4.2. The expressive power of the formalism is limited to facilitate effective and efficient computation of these operations. Due to this limitation, the call-success semantics of a CLP program is usually not expressible in the specification language and the specifications provided for programs describe approximations of the semantics. The approximations will be called *types* following the terminology used in the descriptive approach to types in logic programming.

Our formalism is a generalisation and adaptation to CLP of the ideas of [4.11]. In particular, we add a possibility to distinguish sets of ground (constrained) terms from those containing also non-ground terms.

Types will be denoted by *type terms* built of *type constructors*. Nullary type constructors are called *type constants*.

Some standard type constants are *nat*, *neg*, *any* and *anyfd*:

- *nat* denotes the set $\{0, 1, 2, \dots\}$ of natural numbers,
- *neg* denotes the set $\{-1, -2, \dots\}$ of negative integers,
- *any* denotes the set of all constrained terms with satisfiable constraints,
- *anyfd* denotes the set of constrained terms of the form $c \,[]\, x$, where either x is a variable and c is a constraint describing a finite set of natural numbers, or x is a natural number. (Remember that we do not distinguish between $true \,[]\, x$ and x). This type represents domain variables of CLP(FD) together with their instances.

The remaining types are defined by grammatical rules. For example, the rules

$$p \to 0$$
$$p \to s(p)$$

mean that the type constant p denotes the set of terms $\{0, s(0), s(s(0)), \dots\}$. The meaning of p is formally defined as the set of all terms not containing type symbols that can be derived from p by applying the grammatical rules. For a precise definition the reader is referred to [4.17].

Type *int* of integer numbers may be expressed as union of *neg* and *nat*:

$$int \to neg$$
$$int \to nat$$

Parametric rules with non-ground type terms are also allowed. To apply such a rule, one has to substitute ground type terms for type variables. For example, lists are defined by the following parametric rules:

$$list(\alpha) \to [\,]$$
$$list(\alpha) \to [\alpha | list(\alpha)]$$

Substituting *int* for α gives grammatical rules defining the type $list(int)$, of integer lists. Substituting *anyfd* for α gives rules defining the type $list(anyfd)$ of lists with elements whose type is *anyfd*. The following example shows such a list and illustrates how grammar rules are applied to constrained terms. From $list(anyfd)$ one can derive $[anyfd|list(anyfd)]$ by applying the second rule once. According to the former definition, from the standard symbol *anyfd* one can derive, for instance, $X{\in}\{1, 5, 7\} \,[]\, X$. Hence from $[anyfd|list(anyfd)]$ one can obtain $X{\in}\{1, 5, 7\} \,[]\, [X|list(anyfd)]$ and, in further five steps, $X{\in}\{1, 5, 7\}, Y{\in}\{2..6\} \,[]\, [X, 3, Y]$. The latter constrained term belongs to type *list(anyfd)*, as it has been generated from *list(anyfd)* and does not contain type symbols.

More precisely, a grammatical rule defining an n-ary type constructor t ($n \geqslant 0$) is an expression of the form:

$$t(\alpha_1, \ldots, \alpha_n) \to f(\tau_1, \ldots, \tau_k)$$

where $\alpha_1, \ldots, \alpha_n$ are distinct type variables, f is a standard type constant ($k = 0$) or f is a function symbol of the object language ($k \geqslant 0$), and each τ_i ($1 \leqslant i \leqslant k$) is:

- a type constant or
- a type variable from the set $\{\alpha_1, \ldots, \alpha_n\}$, or
- type terms of the form $t_i(\alpha_{i_1}, \ldots, \alpha_{i_l})$, where t_i is a l-ary type constructor and $\{\alpha_{i_1}, \ldots, \alpha_{i_l}\} \subseteq \{\alpha_1, \ldots, \alpha_n\}$.[3]

In addition, it is required that no function symbol is a principal symbol of two distinct grammar rules defining the same type constructor. Here by a principal symbol of a rule we mean such f that $\ldots [] f(\ldots)$ can be generated from the right hand side of the rule. A finite set of grammar rules satisfying the latter condition will be called a (parametric) *term grammar*.[4]

The types are communicated between the user and the tool in the form of type terms. They refer to a fixed library of grammar rules and to additional rules declared by the user. These rules describe, respectively, the standard types of the tool and the types the user expects to be useful. The analyser generates new rules, if the present ones are insufficient to describe the results of the analysis.

In our approach, the call-success semantics of the program is approximated by providing for each predicate a *call type* and a *success type*; they are supersets of, respectively, the set of calls and the set of successes of this predicate.

4.3 An Example Diagnosis Session

This section gives an informal introduction to our diagnosis technique by demonstrating the use of our diagnoser on an example.

The input of our tool is a CHIP program augmented with an entry declaration specifying a class of initial goals. The result of an interactive diagnosis session is a specification describing the intended types of program predicates and error messages locating clauses responsible for incorrectness of the program w.r.t. this specification.

[3] This restriction on the form of τ_i is added for technical reasons. It implies, informally speaking, that no type is defined in terms of an infinite set of types. For a more general form of this condition see [4.17]. Similar restrictions appear in other approaches to type analysis (e.g. [4.11], the restriction formulated informally).

[4] This extends a traditional notion of regular term grammar, see e.g. [4.11], by using type constants that can introduce constrained terms.

We will demonstrate the use of our diagnoser on the following erroneous n-queens program. The problem being solved by the program is to place n chess queens on an $n \times n$ chess board, so that they do not attack each other. A solution to the problem is represented as a list of length n, where the value j at the i-th position means that the queen on column i is placed on row j. The error is the miss-print in the recursive definition of `safe/3` where the erroneous call `safe(T,X,K1)` appears instead of `safe(X,T,K1)`.

```
:-entry nqueens(nat,any).

nqueens(N,List):-
        length(List,N),
        List::1..N,
        constrain_queens(List),
        labeling(List,0,most_constrained,indomain).

constrain_queens([X|Y]):-
        safe(X,Y,1),
        constrain_queens(Y).
constrain_queens([]).

safe(X,[Y|T],K):-
        noattack(X,Y,K),
        K1 is K+1,
        safe(T,X,K1).
safe(_,[],_).

noattack(X,Y,K):-
        X #\= Y,
        Y #\= X + K,
        X #\= Y + K.
```

The `:-entry` declaration indicates that the predicate `nqueens/2` should be called with a natural number as the first argument (size of the chess board) and *any* term as the second argument. This includes the special case of a variable as the second argument, which however cannot be stated separately in our specification language.

The tool starts its work by computing call- and success-types of every predicate in the program. While doing this, the system informs us that the finite domain constraint `#\=` will be possibly called with incorrect types:

```
Illegal call-type of: X #\= Y
in clause (lines: 19 - 23)

noattack(X,Y,K) :-
        X #\= Y,
```

```
Y #\= X + K,
X #\= Y + K.
```

This kind of warning often coincides with a run-time error. The computed type of the first call of `#\=` is

```
Call-Type: t41 #\= anyfd
t41 --> anyfd
t41 --> []
t41 --> [anyfd|list(anyfd)]
```

and is not a subset of the specified call-type of `#\=` (where the type of the first argument is `anyfd`). Any calls of `#\=` outside of this type result in a run-time error.

In CHIP, the built-in predicate `is/2` is subject to delay until its second argument is ground. In our example the analyser finds that `K` is an integer at the call of `K1 is K+1`, hence this call is not delayed. The treatment of delays in our system is discussed in Section 4.5.

The inferred types can be now inspected by the user. The diagnosis should be started if some of the computed types do not correspond to the user's expectations. In our example, the inspection of the main predicate shows:

```
Call-Type: nqueens(nat,any)
Succ-Type: nqueens(nat,t67)
t50-->[]
t67-->[nat|t50]
```

The call type comes from the entry declaration and the computed success type is described by a term grammar. The intended result should be a placement of n queens on the $n \times n$ chess board, represented by a list of natural numbers. So the expected success type of the second argument is $list(nat)$ (as defined by the grammar rules in Section 4.2.2 with α set to nat) and not the type constructed, which denotes a singleton list of natural numbers. We request diagnosis of the inspected predicate. In response, the diagnoser finds all predicates which may influence the types of this predicate, and asks the user about their intended call- and success-types.

In our example, the diagnoser will request specification of the following types, where `C` stands for "call-type" and `S` for "success-type": `(C)constrain_queens/1, (C)safe/3, (S)safe/3, (S)constrain_queens/1, (S)noattack/3, (C)noattack/3, (S)nqueens/2`.

The types are to be specified one-by-one in arbitrary order. The diagnoser uses this input for generating an error message locating an erroneous fragment of the program. The diagnosis procedure will be explained and justified in the next Section. As already mentioned, the error message is generated as soon as the set of already specified types makes it possible. It may not be necessary to specify all requested types.

A type may be specified either by accepting as a specification the corresponding type constructed by the analyser or by providing a specification different from the latter.

In our example session we follow the specification order suggested by the diagnoser, and we provide one-by-one the following specifications that reflect the rationale behind the program. For instance, the predicate `safe(Q,Qs,D)` describes a relation between a queen placed on a column `Q` (an argument of type *anyfd*) and queens that occupy columns `Qs` (an argument of type *list*(*anyfd*)) situated on the right of `Q`. The parameter `D` (of type *int*) is the distance between `Q` and the first column of `Qs`. The predicate `safe/3` only sets constraints, and therefore its call and success types are the same.

- `(C)constrain_queens/1`: accept the computed call type
  `constrain_queens(list(anyfd))`
- `(C)safe/3`: new specification
  `safe(anyfd, list(anyfd), int)`
- `(S)safe/3`: new specification
  `safe(anyfd, list(anyfd), int)`
- `(S)constrain_queens/1`: new specification
  `constrain_queens(list(anyfd))`
- `(S)noattack/3`: accept the computed success type
  `noattack(anyfd, anyfd, int)`

After providing the last specification we obtain the message telling that the clause

```
safe(X,[Y|T],K) :-
        noattack(X,Y,K),
        K1 is K+1,
        safe(T,X,K1).
```

violates the specification (see also Fig. 4.3).

Intuitively, from the specification of the success type of `noattack/3` we get the type `anyfd` for the variable `X`, whereas on the call of `safe/3` this variable is expected to be of the type `list(anyfd)`. A similar clash appears in the case of variable `T`.

Figures 4.2 and 4.3 show the graphical user interface of the diagnoser. It has three information windows and a display. After completing the analysis phase the leftmost window shows all predicates of the program. The computed call- and success- types of a predicate can be displayed by clicking a predicate in this window. If they do not conform to the user's expectation, diagnosis may be started by pressing the button "Diagnose". In response, the diagnoser generates the list of all types which may be needed to be specified for completing the diagnosis and displays them in the "Ask" window. At each step of the diagnosis session this window shows which types remain to be specified. The "User" window shows which types have been already specified during the session. In the "Ask" window one selects a type to be

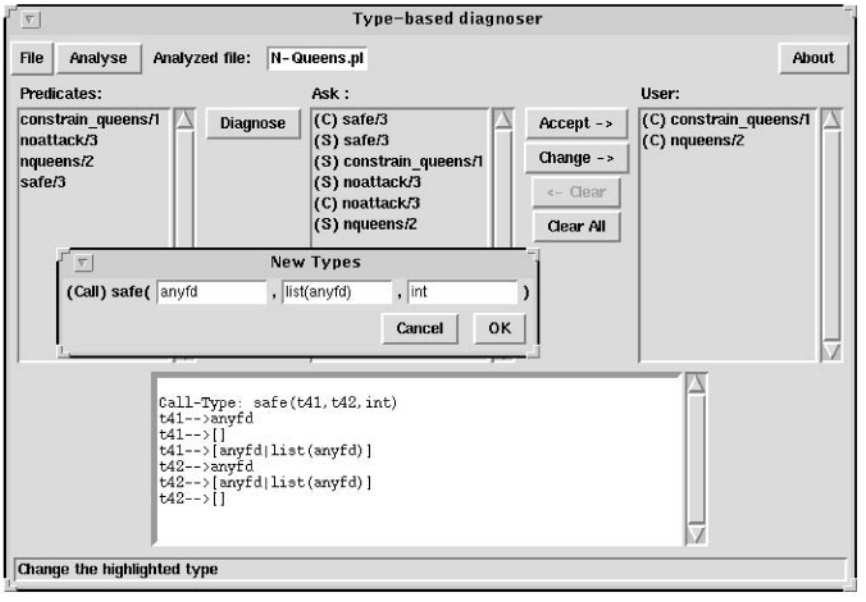


Fig. 4.2. Specifying a new call type for `safe/3`.

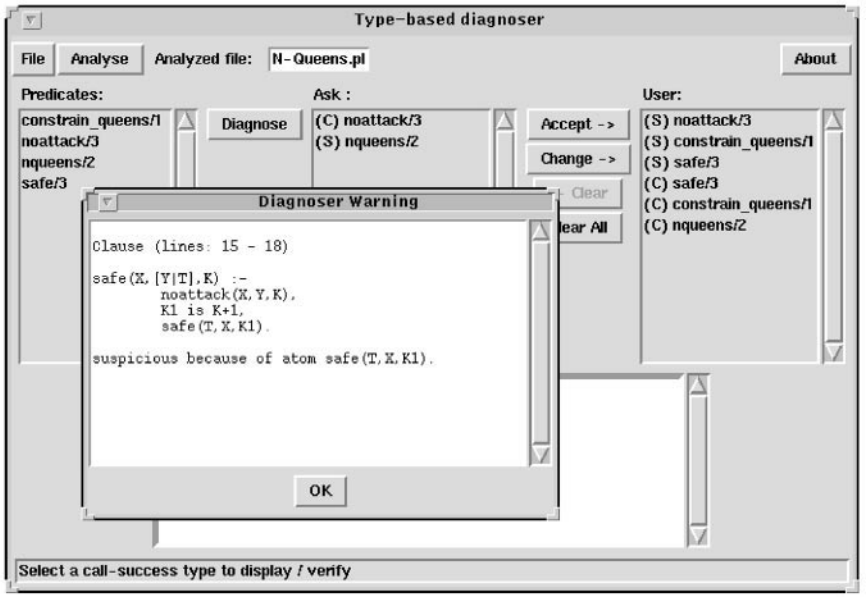


Fig. 4.3. A warning given by the diagnoser.

specified. The corresponding type inferred by the analyser is shown in the display at the bottom. The user may accept it as a part of the specification, by pressing the "Accept" button. Otherwise, clicking the "Change" button will trigger a pop-up window for writing a new specification for the active entry of the "Ask" window, as illustrated in Figure 4.2. The already specified types shown in the "User" window may be inspected and withdrawn (by pressing the button "Clear" or "Clear All"), if the user decides to specify them differently. Figure 4.3 shows the warning generated by the diagnoser.

The number of the requested types to be specified by the user in a diagnosis session depends on the predicate for which the diagnosis was initiated. This number is often rather small. However, in our example the request concerned the main predicate so the initial "Ask" list included the types of all the predicates in the program (except for built-ins, whose types have been retrieved from a library, and the call-type of the main predicate).

Observe that the diagnosis engine is a compile time tool. It does not require performing any test computations of the diagnosed program. In contrast to the usual debugging techniques, like tracing or even declarative debugging, it is not driven by error symptoms appearing in test computations. The diagnoser has access only to the diagnosed program and to its specification constructed interactively by the user with the help of the results of static analysis. For finding the error it uses partial correctness proof techniques. As the example shows, an error may be sometimes located by providing only part of the specification. The role of the program analyser is auxiliary and in principle diagnosis can be done without it. However, in our tool this option has not been implemented and the analyser is invoked at the beginning of each session. Knowing a difference between a type found by the analyser and that expected by the user, makes it possible to focus the diagnosis to a fragment of the program. The part of the analyser's output that is found by the user to be correct, is re-used as a (part of) the specification.

In our example, the diagnosis was started because the success type of `nqueens/2` computed by the analyser did not agree with the user's expectations. Notice that it was a proper subset of the expected type. This suggests *incompleteness*, i.e. the fact that some of the expected answers cannot be computed by the program. An incomplete program can still be partially correct w.r.t. to the specification considered. However, an error in a program often causes both incorrectness and incompleteness. Luckily, this was the case in the example even though the incorrectness was not visible in the type of the main predicate found by the analysis.

4.4 The Diagnosis Method

Now we describe the principles of our diagnosis approach. As already mentioned, to simplify the presentation we assume here that programs are executed under the Prolog selection rule (i.e. without delays).

4.4.1 Correct and Incorrect Clauses

The purpose of diagnosis is to locate in a program all the fragments responsible for its behaviour being incorrect w.r.t. a given specification. They should be as small as possible. The erroneous fragments located in our approach are program clauses, more precisely *clause prefixes*. A prefix of a clause $H \leftarrow B_1, \ldots, B_n$ is any formula $H \leftarrow B_1, \ldots, B_k$, where $k \leqslant n$.

We have to define precisely what an incorrect clause (prefix) is. It has to be *the* reason that some call or success in the computations violates the specification. The idea is that a clause is considered incorrect if it leads to an incorrect call or success, despite the rest of the program behaving correctly.

Example 4.4.1. We show an example of a clause that leads to an incorrect call even if the rest of the program behaves correctly. Consider the clause located by our diagnoser in the example session of Section 4.3.

```
safe(X, [Y|T], K) :-
        noattack(X, Y, K),
        K1 is K + 1,
        safe(T, X, K1).
```

Assume that the rest of the program behaves correctly: each success of a predicate is in its specified success-type. With this assumption we examine the clause. The specification is that given to the system in the example diagnosis session of Section 4.3. Additionally, the call type of `noattack/3` is the same as its success type.

1. Assume that the clause is resolved with a call that belongs to the specified call-type `safe(anyfd,list(anyfd),int)`. Hence, the subsequent call to `noattack/3` must belong to `noattack(anyfd,anyfd,int)`; this follows from the structure of arguments in the head of the clause and from the definition of the type `list(anyfd)`. This conforms to the specified call-type of `noattack/3`, the obtained type is a subset of the specified call-type of `noattack/3`.
2. Assume that the call to `noattack/3` succeeds and the success of this call is in the specified success-type `noattack(anyfd,anyfd,int)`. Hence the built-in `is/2` is called according to its specification (which requires that the second argument is a ground arithmetic expression).
3. Consequently the success of `is/2` is in its specified success type (`K1` is bound to an integer). Hence the subsequent call to `safe/3` is in the type `safe(list(anyfd),anyfd,int)`. This is in conflict with the specified call-type of `safe`, which is `safe(anyfd,list(anyfd),int)`. The former is not a subset of the later.

Thus, even if the rest of the program behaves correctly, the clause leads to an incorrect call of `safe`. This illustrates our idea of incorrect clause. An incorrect prefix is the fragment of the clause beginning with its head and ending with the atom, of which an incorrect call was found.

Similarly, a clause may lead to a success that is not in the success-type of the head predicate. For example, the clause `p(X) :- q(Y)` with the specification `(C) p(any), (S) p(int), (C) q(any), (S) q(int)` is incorrect since even if the rest of the program behaves as specified this clause may lead to a success that is not in the specified success type `p(int)`.

Now we describe the concept of incorrect clause in a more formal way. It is defined w.r.t. to a fixed specification (which includes call- and success types of constraint predicates and built-ins). Below, by a correct call or success we mean a call or success correct w.r.t. this specification. Consider a clause

$$C \;=\; p_0(\boldsymbol{u}_0) \leftarrow p_1(\boldsymbol{u}_1), \ldots, p_n(\boldsymbol{u}_n).$$

Assume that at some step of derivation a correct call $c \,[]\, p_0(\boldsymbol{t}\,)$ appears and clause C is used. Assume that then (some instances of) $p_1(\boldsymbol{u}_1), \ldots, p_k(\boldsymbol{u}_k)$ become current calls ($0 \leqslant k \leqslant n$). Let us denote these calls by $A_1, \ldots, A_k$. Assume that the corresponding successes $A'_1, \ldots, A'_{k-1}$ are correct. If the call A_k is incorrect (for some choice of a correct call $c \,[]\, p_0(\boldsymbol{t})$ and correct successes $A'_1, \ldots, A'_{k-1}$) then C is considered an *incorrect clause*. Moreover the prefix $p_0(\boldsymbol{u}_0) \leftarrow p_1(\boldsymbol{u}_1), \ldots, p_k(\boldsymbol{u}_k)$ of C is said to be incorrect.

Similarly, if $k = n$ and all the calls $A_1, \ldots, A_n$ succeed then the first call $c \,[]\, p_0(\boldsymbol{t}\,)$ also succeeds. If the latter success is incorrect, for some choice of a correct call $c \,[]\, p_0(\boldsymbol{t})$ and correct successes $A'_1, \ldots, A'_{k-1}$, then C is an *incorrect clause*. (We also say that C is an incorrect prefix).

It can be proved that if the program P with a class of initial atomic goals $\mathcal{G}$ is incorrect then it contains an incorrect clause, provided the goals in $\mathcal{G}$ are correct. (Remember that we assume a fixed specification). Thus showing that there are no incorrect clauses means proving that the program is correct.

The reverse of this property does not hold. A correct program may contain a clause which is incorrect according to the definition above. Roughly speaking, the reason is too weak a specification. There may exist incorrect clauses in the program despite the program (together with a set of correct goals) being correct. This is related to the fact that the specification for a correct program is an over-approximation of the real call-success behaviour. The notion of incorrect clause refers to the specification, and the calls and successes obtained by the incorrect clause from the specification may not appear in the computation of the program. Notice however that it is still justified to call such a clause incorrect. When placed in another program, with the same specification for the common predicates, the clause can be a reason of program incorrectness.

Example 4.4.2. Here we show a correct program containing an incorrect clause. The reason is that the specification is too weak. Moreover, a sufficiently strong specification does not exist (in the considered class of specifications).

Consider a clause

```
p( X ) :- prime( N ), q( N, X ).
```

Assume that `prime` succeeds always with a prime number and that `q(N,X)` succeeds with `X` bound to a natural number if `N` is prime, and to a negative integer if `N` is not prime. Assume that the specification requires that `p` succeeds with a natural number.

The set of prime numbers cannot be expressed as a type. The best specification we can have for the success-type of the argument of `prime`, and for the call-type of the first argument of `q`, is the set of natural numbers `nat`. For the clause to be correct, the success-type of the second argument of `q` has to be `nat` (or its subset). With such a specification however, we obtain incorrectness of the part of the program defining `q`.

So we can construct specifications such that the success type of `p` is `p(nat)` and the program is correct. However for each such specification, either the clause above or some other clause of the program is incorrect.

Example 4.4.3. Here we show that a problem of an incorrect clause occurring in a correct program can be caused by different usages of a procedure in the program.

Consider the example from Section 4.3 and a (corrected) clause

```
safe(X,[Y|T],K) :-
        noattack(X,Y,K),
        K1 is K+1,
        safe(X,T,K1).
```

This clause is correct w.r.t. the specification discussed in that section. Let us change the specification, replacing `int` by `nat` in the call and success types of `safe`. So both these types are now `safe(anyfd,list(anyfd),nat)`. The program is still correct for this specification, the last argument of `safe` is never a negative integer. However, the clause becomes incorrect. The success type of the first argument of `is` is specified to be `int`. It cannot be any smaller type. (Imagine that `is` appears somewhere else in the program and there negative integers may result too). But now the clause is incorrect, as the call type of the third argument of `safe(X,T,K1)` we obtain `int`, which is not a subset of `nat`.

To solve the problem illustrated by the last example, one would need different type specifications for different occurrences of a predicate.

Summarising, our diagnosis method consists in finding all the incorrect clause prefixes (in a given program, w.r.t. a given specification). This means locating all the errors in the program. Maybe some located prefixes are incorrect only due to a weakness of the specification, as discussed above, and are not reasons for actual program incorrectness. However *all* the actual reasons for the program being incorrect are within the located prefixes. In particular, when no incorrect clause is found then the program is correct, provided the initial goals are correct.

4.4.2 Incorrectness Diagnosis

Incorrectness diagnosis is performed by automatic identification of clauses that are incorrect w.r.t. to a given specification. The core of the diagnoser is thus an algorithm that automatically checks correctness of prefixes of a clause. The idea is to use the specification for computing a superset of all possible calls of the last atom of a given prefix (and of all successes of the head if the prefix is the complete clause). Correctness of the prefix is established by checking inclusion of the result in the set defined by the call specification of this atom (or by the success specification of the head). The correctness check may be described in terms of some primitive operations. We describe them referring to the example:

- *Deconstruction.* This operation determines the type of a subterm of a given typed term. Consider the clause of Example 4.4.1. The call type of its head predicate is `safe(anyfd,list(anyfd),int)`. The head is `safe(X,[Y|T],K)`. Using the deconstruction operation we obtain types of the terms bound to the variables in the head in any correct call. For `Y` and `T` it yields types `anyfd` and `list(anyfd)`, as the type of `[Y|T]` is `list(anyfd)`. (The types of `X` and `K` are trivially `anyfd` and `int`). The reader may check that deconstruction is easy to compute using the rules defining types.
- *Type intersection.* Suppose that, by means of deconstruction, types are found for some occurrences of a variable. Then its possible values are determined by the intersection of the types. Assume that the example clause has been called as specified and that the execution reached the call of `is/2`. The variable `K` at this point is bound to a constrained term that is in the intersection of the call-type of the third argument `safe/3` and the success type of the third argument of `noattack/3`. Both types are `int`, so the intersection is trivial in this case. The algorithm used in our tool for computing intersection of types is described in [4.15, 4.29].
- *Construction.* This operation determines the type of a term from types of its subterms. Usually, new grammar rules have to be constructed to describe it. In our example, the type, say `t10`, of `K+1` can be computed knowing the type of `K` (`nat`) and `1`:

```
t10 --> nat + t11
t11 --> 1
```

- *Type inclusion.* To establish (in)correctness of a clause we have to check whether the clause leads to (in)correct calls or successes. For example, whenever the rest of the program behaves as specified, the first argument of the last body atom will be bound at call to a term in `list(anyfd)`. This has been established by deconstruction of the initial call. The specified call-type for this argument is `anyfd`. Incorrectness is established by checking that the former type is not a subset of the latter. The algorithm used in our tool for checking type inclusion is described in [4.15, 4.29]

The algorithm checking whether a clause can generate incorrect calls can now be outlined as follows:

For each body atom $B = p(...)$,
  for each variable X in B,
    for each occurrence i of X in the preceding atoms,
      compute its type t_i
      (by type deconstruction, from the specified
      call type of the head or, respectively,
      success type of a body atom).
    The type of X at B is the intersection of all t_i's
    (it is "any" if there are no such occurrences).
  $type(B)$ is determined by type construction
    from the types of its variables.
  If $type(B)$ is not a subset of the call type of p
    then the clause is incorrect.

The computed $type(B)$ is the set of all calls of B that would appear in the computations starting by correct calls of the clause, provided that the success set of any body atom coincides with the specified success type of its predicate.

A similar algorithm checks whether a clause can generate incorrect successes. One just takes the clause head as B and considers all the occurrences of X in the whole clause. The obtained type of B is then checked to be a subset of the success type of p.

If the specification at hand describes all predicates of the program the above algorithms can be used to find all incorrect clauses. If some type specifications are missing, we can use the following technique. We describe it for the case where $B = p(...)$ is a body atom (and under assumption that the specification of the call type of B is not missing).

First replace all the missing types by the most general type *any*. Now the computed $type(B)$ includes all the calls of B, for any possible specification of the missing types. If $type(B)$ is a subset of the specified call type of p then no incorrectness related to B may occur. The last condition of the algorithm is independent from the missing types. Conversely, suppose that after replacing the missing types by the empty type the computed call-type of B is not a subset of the specified one. Then the clause is incorrect.

4.5 Delays

Modern Prolog and CLP implementations use Prolog selection rule with delays. This means that, for some predicates, an atom can be selected only if its arguments are sufficiently instantiated. The requirements on the form of selectable goals are given by delay declarations. We assume that the set of

selectable goals can be described in our type formalism. In this section we explain handling of delays by our system. The presentation is informal. The predicates subject to delays will be called *delayed* predicates. For simplicity, we deal only with built in delayed predicates.

The approach is based on a simple but rather imprecise approximation of the semantics of delays. The main idea is to deduce, whenever possible, that a predicate call will not be blocked (i.e. will be selected by the Prolog selection rule). Then we treat it as described in the previous section. Otherwise we do not conclude anything about when the call will be actually selected.

In the context of delays we have an additional notion of an initial procedure call. By an *initial call* (of procedure p) we mean a constrained atom $c \,[]\, p(\dots)$ such that $p(\dots)$ is selected by the Prolog selection rule in a goal with constraints c. If the atom is selectable then it is actually selected. Otherwise it is *blocked*. By an *(actual) call* we mean a constrained atom $c \,[]\, p(\dots)$ such that $p(\dots)$ is actually selected (in a goal with constraints c).

Also the notion of success is different. It is convenient to treat a (sub)goal as succeeding also when some blocked atomic goals remain unresolved. To describe this notion of success more precisely, assume that all the unifications in the resolution are variable renamings. Consider a derivation starting from a goal $G_0 = A_1, \dots, A_n$. Assume that the first atom of G_0 is actually selected. Consider the first goal in the derivation containing atoms $A_2, \dots, A_n$ and possibly some blocked, non selectable atoms. Let c be the constraint of this goal. Then $c \,[]\, A_1$ is the *partial success* of A_1 in this derivation. So for computations without delays partial successes coincide with successes. If a constrained atom is blocked then it itself is its partial success. Now we assume that a success-type specifies (a superset of) the set of partial successes of the selectable calls of the respective predicate.

As discussed in Section 4.2.1, we do not treat constraint predicates as delayed. Whenever an atomic constraint is initially selected, it is added to the constraint store. Thus it immediately succeeds or fails depending on the result of the constraint consistency check.

For each delayed predicate, instead of its call-type, we maintain its selection-type describing the set of selectable calls. If an initial call is in this set then it is immediately selected (i.e. not delayed)[5]. For instance, the selection-type for the built-in predicate `is/2` of CHIP is the set of constrained atoms $c \,[]\, (u \,\texttt{is}\, w)$ where w is a ground arithmetic expression (and c, u are arbitrary).

Now we present an algorithm for checking correctness of a clause in the context of derivations with delays. The algorithm is a modification of that shown in Section 4.4.2. Given a clause $H \leftarrow B_1, \dots, B_n$, the algorithm approximates the set of initial calls of each B_i. If the result is a subset of the

[5] So if we want to approximate the set of the selectable calls, the selection-type has to be a subset of this set, not a superset.

selection-type of B_i then we are sure that B_i is not blocked (even if its predicate is a delayed one).

Mark { as not blocked } the head H and the body atoms with non delayed predicates.
For $i = 1, \ldots, n$
  { Compute what happens before the initial call of B_i }
  For each variable X occurring in B_i,
    for each occurrence x of X in a marked atom from $H, B_1, \ldots, B_{i-1}$
      compute its type t_x using type deconstruction operation
      (from the specified call type of H or, respectively,
      the success type of B_j).
    Compute the intersection t_X of all t_x's.
    { Type t_X is an approximation of the set of values of X at the initial call of B_i }
  Compute an approximation *type*(B_i) of the set of initial calls of B_i,
  from the types t_X of its variables, by the type construction operations.
  If *type*(B_i) is a subset of the call/selection-type of B_i then
    { B_i is not blocked and the clause prefix $H \leftarrow B_0, \ldots, B_i$ is correct }
    mark B_i.
  Else
    if B_i is a call of a non delayed predicate then
      { the prefix may be incorrect }
      signal a warning,
    else
      { B_i may be blocked. }
      If the intersection of *type*(B_i) and the call/selection-type of B_i is empty then
        { B_i will never be selected }
        signal a warning,
      else
        { the clause prefix $H \leftarrow B_0, \ldots, B_i$ is correct }

A similar, second part of the algorithm checks that the clause cannot generate incorrect successes of H.

If the algorithm does not find any incorrect clause prefix in a program P then P is correct in the following sense. In any computation of P starting from an atomic goal which is in its call/selection-type,

- any (actual) call is in its call/selection-type,
- if an atom has been marked as not blocked then any its call is not blocked (any its initial call is an actual call),
- any partial success of a non blocked call is in its success type.

The same property holds if the only warnings issued by the algorithm are related to the last check in the algorithm (B_i never selected). Such a warning may correspond to a run-time error. As an example take B_i to be ...`is`... and suppose that no atom in *type*(B_i) has an arithmetical expression as the second argument, so that the intersection considered in the check is empty.

Then execution of the program by CHIP will stop with a run-time error (unless B_i is never initially selected).

Starting from the algorithm described above it is rather obvious how to generalize our type analysis algorithm for programs with delays.

Example 4.5.1. In the scalar product program below, each call of is/2 is blocked, as the second argument is not ground.

```
:- entry sp(any,list(int),list(int)).

sp(N1,[P|T],[Q|R]) :- N1 is P*Q+N, sp(N,T,R).
sp(0,[],[]).
```

This program is found correct when both the call- and success-type of `sp` are specified to be `sp(any,list(int),list(int))`. Also program analysis finds these types. Actually, the program is correct also for the success-type `sp(int,list(int),list(int))`, as the blocked calls of `is` are eventually selected and `sp` succeeds with ground first argument. Our algorithm is however too weak to determine the correctness of P for this stronger specification.

The described approach is applicable to any kind of delayed predicates, not only to built-ins, but in the current version of the tool delay declarations are not yet supported.

An advantage of the approach is its simplicity and ease of augmenting the system to handle delays. Its generalizations are a subject of future work. Existing static analysis techniques of delays (see e.g. [4.22] and references therein) are potentially interesting in this context. Applying them for our purposes may however require specifications which describe more than just call- and success-types. We expect that a major improvement in our approach can be achieved by discovering the cases when a blocked body atom is selected before "the clause succeeds". It seems that it is possible to extend for this purpose the dependency technique of directional types discussed in [4.2].

4.6 The Diagnosis Tool

This section surveys the main design decisions of the existing prototype implementation of our tool. Its main components are the analyser that computes types (which approximate the actual semantics of a given program) and the diagnoser locating erroneous clauses. They have some common parts. We begin the section by explaining the usage of parametric and parameterless grammars by the tool. Then we describe its two main parts.

Both the analysis and diagnosis algorithms work with types represented as parameterless regular term grammars. Equivalently, such grammars can be seen as a restricted class of CLP programs [4.15]. Parametric grammars are employed only in the user interface. There is a library of type definitions

which may be augmented by the user. It contains for instance a parametric grammar defining type $list(\alpha)$ (Cf. Section 4.2.2). Whenever possible, the types computed by the system are presented to the user in terms of those defined in the library or declared by the user. In this way the user faces familiar and meaningful type names instead of artificial ones. For instance, assume that the system has to display a type $t77$ together with grammar rules $t77 \rightarrow [\,]$, $t77 \rightarrow [t78|t77]$. It finds that they are an instance of the rules defining $list(\alpha)$ and displays $list(t78)$ instead. Similarly, the user's input can refer to the types defined in the library.

The principles of the analysis are described in a separate paper [4.16]. The analyser and the diagnoser share some basic components. The algorithm is implemented in SICStus Prolog [4.30], the implementation is based on that by [4.20]. We made several lower level improvements to the original implementation, like introducing more efficient data structures (AVL trees instead of lists). They resulted in substantial improvement in analysis efficiency. Running the analyser on a number of examples, we have observed speedup with an approximate factor 4.

The type analysis algorithm constructs call and success types of the predicates defined by program clauses, thus computing an approximation of their call-success semantics. To be able to deal with real programs, it uses a library of type specifications of built-in predicates. Similarly it is able to deal with fragments of programs (for instance with programs under development). In the latter case the user is required to provide type descriptions for the undefined predicates. Such descriptions correspond to `trust` assertions of Chapter 1.

The types constructed by the analyser are on request shown to the user, who may decide to start diagnosis, as illustrated in Section 4.3. The diagnosis relies on the type specification provided incrementally by the user. As discussed in Section 4.3, the specification process is supported by the possibility to accept some types constructed in the analysis phase as specified ones. It is also restricted to the predicates relevant for the diagnosed predicate. Moreover, a heuristics is used to suggest to the user the order of specifying types. The suggestion is reflected by the order of requests in the "ask" window. Following this order often results in fewer type specifications needed to locate an error. The user may stop the diagnosis with the first error message, which is often obtained without specifying all requested types. The diagnosis process may be continued by specifying all requested types. In this case, the tool will locate all incorrect clause prefixes in the fragment of the program relevant for the diagnosed predicate.

An error message contains an incorrect clause. Its incorrect prefix is indicated by referring to the atom whose type computed by the method of Section 4.4 is not a subset of the respective specified type. This atom is the head of C or the last atom of an incorrect prefix.

The specification provided by the user is stored by the diagnoser and may be re-used during further diagnosis sessions.

4.7 Limitations of the Approach

Our approach can be characterized by several design decisions that we have made, such as the semantics considered, the specification language and the diagnosis method. As always in such cases, there is a trade-off between efficiency and precision. Therefore, some limitations of the framework were inevitable. This section surveys them.

We discuss the restricted expressive power of type specifications, a need for incompleteness diagnosis and a need to have more than one pair of types for a predicate. We also show an example where the call-success semantics is inappropriate to express programmer intuitions.

The restricted expressive power of our specification language makes it possible to perform effective analysis and diagnosis of programs but permits to detect only the errors that violate specifications expressible in this language. This is illustrated by the following example.

Example 4.7.1. Let us consider the following erroneous *append* program, where variables `Xs` and `Ys` are swapped in the clause body.

```
:- entry app(list(int),list(int),any).

app([], Xs, Xs).
app([X|Xs],Ys,[X|Zs]) :-
   app(Ys,Xs,Zs).
```

What we obtain if we invoke the call-success diagnoser with *intended* success-type `app(list(int),list(int),list(int))` is that there are no incorrect clauses. Hence, the program is correct w.r.t. the intended specification. This kind of error cannot be detected by a type based approach because, informally speaking, the type of the two variables `Xs` and `Ys` is the same.

For some built-in predicates, the restricted expressive power of types makes it impossible to express the form of the allowed calls. Thus our approach is not able to discover some of the errors of an incorrect call of a built-in.

We illustrate the problem of lack of the incompleteness diagnosis in our tool by the following example.

Example 4.7.2. Consider an erroneous version of a program for "closing" open (or partial) lists.

```
:-entry close_list(any).

close_list([]).
close_list([_|Xs]):-
    close_list(xs).
```

The recursive call of `close_list/1` has the constant `xs` as its argument instead of the variable `Xs`. The type inferred by the analyser is:

```
Succ-Type: close_list(t15)
t15-->[]
```

It is obvious that some values (i.e. all the non-empty lists) cannot be computed by the program. That means an incompleteness symptom. Assume now that the programmer has completed the specification with the success type `close_list(list(any))`. Observe that, as the second clause always fails, the only success of `close_list/1` (i.e. `close_list([])`), is contained in the type `close_list(list(any))` and therefore the diagnoser will give us no warning. Thus the program is correct w.r.t. the specification, although it is incomplete.

Next, we show that having only one call and one success type per predicate is actually a restriction.

Example 4.7.3. The program below deletes an integer number from a given list of integers, by means of the `app/3` predicate.

```
:-entry del(int,list(int),any).

del(E,L,L0) :-
    app(L1,[E|L2],L),
    app(L1,L2,L0).

app([],Xs,Xs).
app([X|Xs],Ys,[X|Zs]):-
    app(Xs,Ys,Zs).
```

Observe, that in the clause defining `del/2` the predicate `app/3` is used in two ways: first to decompose the list `L` and then to concatenate `L1` and `L2`. The types inferred by the analyser are:

```
Call-Type: del(int,list(int),any)
Succ-Type: del(int,list(int),any)

Call-Type: app(any,any,any)
Succ-Type: app(list(any),any,any)
```

The call type of `app/3` is so general because it has been computed taking into account all three usages of this predicate in the program. Call types originating from these three program points have been merged, by means of the upper bound operation. Notice, that no specification (with an entry declaration as above) for which the program remains correct, can contain more accurate call type for `app/3`, as we can specify only one call type per predicate.

Analysing various calls of the same predicate separately would obviously bring more precise results and would require employing a *polyvariant* analysis method, as that of [4.24].

Essentially the same limitation affects the treatment of built-ins. Their success types are described taking into account all their possible usages. As pointed out in Section 4.4 this may make the diagnoser generate undesired warnings (as a clause of a correct program may turn out to be incorrect, due to a too general success type of a built-in).

The call-success semantics may not be suitable, as concerning the use of logical variables. The user may not be interested in the actual calls but rather in the successes related to initial calls. We illustrate this by the example originating from [4.2].

Example 4.7.4. The following program analyses a binary tree `T` with nodes labeled with natural numbers and constructs a binary tree `NT` of the same shape with all nodes labeled with the maximal label of `T`. The program includes a type declaration defining a parametric type `tree(A)` by two grammar rules (conf. Section 4.2.2).

```
:- typedef tree(A) --> void; t(A,tree(A),tree(A)).
:- entry maxtree(tree(nat),any).

maxtree(T,NT) :- maxt(T,Max,Max,NT).

maxt(void,_,0,void).
maxt(t(N,L,R), Max, MaxSoFar, t(Max,NewL,NewR)):-
    maxt(L,Max,MaxL,NewL),
    maxt(R,Max,MaxR,NewR),
    max(N,MaxL,MaxR,MaxSoFar).

max(A,B,C,A) :- A >= B, A >= C.
max(A,B,C,B) :- B >= A, B >= C.
max(A,B,C,C) :- C >= A, C >= B.
```

The call-success analyser infers the following types:

```
Call-Type: maxtree(tree(nat), any)
Succ-Type: maxtree(tree(nat), tree(any))

Call-Type: maxt(tree(nat), any, any, any)
Succ-Type: maxt(tree(nat), any, nat, tree(any))

Call-Type: max(nat, nat, nat, any)
Succ-Type: max(nat, nat, nat, nat)
```

Hence, it correctly shows that during the execution some successes of `maxt` have an argument of the type `tree(any)`, since the constructed trees have nodes labeled by variables. To show that in the final result both arguments of `maxtree` are of the type `tree(int)`, one has to use a richer class of specifications[6] or refer to a different semantics and use different proof methods, like the method shown in [4.2]. A type diagnoser based on that method can be constructed by applying similar approximation techniques to the verification conditions of that method.

4.8 Related Work

Our approach and the design of our tool is a novel contribution, but its components and principles are based on well-known ideas and techniques of logic programming, which, however, require extension and adaptation to CLP.

The question of what the reason of an incorrect behaviour of a program is has been investigated in the framework of declarative diagnosis [4.31, 4.25, 4.18]. The concept of incorrectness error originating from that work can be related to the program not being partially correct w.r.t. a specification. Eventually this can be linked to violation of a verification condition in a proof method for partial correctness of run-time properties. Several such methods has been discussed in the literature, e.g. [4.14, 4.1, 4.16]. Our concept of incorrect clause can be linked to such a verification condition.

The general idea of using semantic approximations for program verification and for locating errors was discussed in our previous work [4.3]. That paper was the main inspiration of this work.

The static diagnosis technique presented here is similar to the abstract diagnosis of [4.8]. The latter is essentially a method for verifying a program against a specification that describes a program property. It is required that the space of possible properties forms a *Galois insertion* with the semantics of the programming language. Our type domain does not fit into the Galois insertion framework due to non existence of an abstraction function [4.15, 4.29]. Unlike our approach, the work of [4.8] is not focused on implementing a tool, but rather on theoretical aspects of the diagnosis problem. It aims at finding both incorrectness and incompleteness errors.

The decision to use term grammars as a specification language made it possible to extend for our purposes the well-known results and techniques on regular sets and regular term grammars and the techniques of constructing regular descriptive types for logic programs. There is a vast literature about it (see e.g. the survey in [4.27]). More specifically, our term grammars can be

[6] We need to express that, at a success of `maxt(T,M,Max,NT)`, `NT` is a tree with all the nodes labeled by `M` and `Max` is an integer.

directly linked to regular term grammars (see e.g. [4.11] and references therein) and to deterministic root-to-frontier tree automata [4.21]. Our grammars provide "ad hoc" extension of the grammars of [4.11], for dealing with CLP over finite domains. In [4.29] we present a more systematic way for extending regular term grammars with constraints, which is however not directly used in our tool. There have been many proposals for constructing descriptive regular types for logic programs, e.g. [4.28, 4.19, 4.24, 4.4]. We adopted for extension to CLP the technique of [4.20]. One of the reasons was that in that case we were able to re-use part of the analyser code for diagnosis.

The paper [4.3] was also a starting point for developing two tools that are strongly related to ours, both are described in this volume.

The first one is a framework of assertion-based debugging (Chapter 2, [4.23]). The assertion language used in that framework is a superset of our specification language. It not only allows to express call- and success-types but also richer properties of calls and successes and several other properties such as determinacy, non-failure, cost, etc. Abstract interpretation is used to infer some properties of the program, including types. (The type inference program has been ported for that purpose from our tool). The user may provide a priori a partial specification by stating some assertions. These are automatically compared with the inferred assertions. The comparison may confirm that the latter imply the former. A failure in verifying this may be due to program error or to incompleteness of the checking method. In the case of such failure the system generates a warning and allows to incorporate run-time checks for non-verified assertions. A special warning is issued when the comparison shows that the inferred assertion and the specified one do not intersect. This implies that the latter will be violated by each computation of the program (or the control will not reach the corresponding program points).

The tool reports all abstract symptoms that can be found with given assertions. It does not explicitly locate the erroneous clauses.

The assertion tool of Prolog IV, described in Chapter 3, uses verification techniques similar to ours for locating erroneous clauses at compile time. However, the assertion language is different from ours. The assertions are built from a fixed number of primitives, mostly some predefined constraints of Prolog IV. This gives less flexibility than our types, but allows direct use of constraint solver for verification of assertions. If an assertion cannot be proved statically then a run-time test is generated, like in the method of Chapter 1 and [4.23]. In contrast to our approach, the specification has to be given a priori and the specification process is not supported by static analysis.

Our work was initially described in [4.6, 4.7]. Since then, the treatment of delays was added and the presentation of the method has been substantially changed and improved. The type description formalism was changed, a graphical user interface was designed and implemented and the efficiency of the analysis has been improved by modification of the analysis algorithm.

4.9 Conclusions and Future Work

By extending to CLP well-known techniques of LP and combining them in an innovative way we constructed an interactive tool that facilitates location of errors in CHIP programs. The errors dealt with are incorrect calls and successes of predicates. The incorrectness is considered with respect to a restricted class of specifications, namely type specifications. The principles of our approach can be summarized as (1) automatic synthesis of types that approximate a program's semantics and are easy to understand by the user, (2) automatic location of the errors and (3) minimizing and facilitating the specification effort. As a side effect of a diagnosis session a specification of the program is obtained which may be used in future diagnosis and for documentation purposes.

In contrast to most of debugging approaches, our tool does not refer to any test computations of the program. Also, the algorithm is able to work without any information about error symptoms. As the class of considered specifications is restricted, many errors are outside the scope of the method. On the other hand, the diagnosis algorithm locates exactly those clauses (and clause prefixes) that are incorrect w.r.t. the specification.

Our approach can be seen as a kind of type checking. However, it does not impose any type discipline on the program, it does not require providing type declarations in advance and often only a part of these declarations is sufficient to locate an error.

The role of static typing in discovering errors at compile time is well known. We believe that the presented tool adds to untyped CLP languages some important advantages of statically typed languages.

As stated in Section 4.8, relevant techniques adopted concern:

- representing and manipulating regular sets [4.11],
- construction of regular approximations of logic programs [4.20],
- methods for proving properties of logic programs [4.14, 4.1, 4.13, 4.16]

We had to extend them to handle constrained terms. The efficiency of type construction by our tool was acceptable on a benchmark consisting of the CHIP demonstration programs, but we hope to be able to improve it. For instance optimizations similar to those suggested in [4.9] should be possible, as our type analysis algorithm is equivalent to bottom-up abstract interpretation of the magic transformation of the source program. Developing a different algorithm for construction of types may be another way of attacking this problem. Extension of the tool for other constraint domains would require addition of new standard types. The topics of future research include: better handling of delays, diagnosis of incompleteness, improving the heuristics of query ordering and introducing parametric polymorphism into specifications[7].

[7] We already can specify parametric types by using type variables in grammatical rules. However non-ground type terms are not allowed in the specifications. Such

References

4.1 A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. Lecture Notes in Computer Science.

4.2 J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, 1997.

4.3 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

4.4 W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proc. of SAS'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

4.5 K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

4.6 M. Comini, W. Drabent, J. Małuszyński, and P. Pietrzak. A type-based diagnoser for CHIP. ESPRIT DiSCiPl deliverable, September 1998.

4.7 M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP programs using type information. In Proceedings of APPIA-GULP-PRODE'99 – 1999 Joint Conference on Declarative Programming, L'Aquila, Italy, 1999. (Preliminary version appeared in Proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98, Manchester, 1998).

4.8 M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–95, 1999.

4.9 M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

4.10 Cosytec SA. *CHIP System Documentation*, 1998.

4.11 P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.

4.12 P. Deransart. Proof method of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

4.13 P. Deransart and J. Małuszyński. A Grammatical View of Logic Programming. The MIT Press, 1993.

4.14 W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

4.15 W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, September 1998.

4.16 W. Drabent and P. Pietrzak. Type Analysis for CHIP. In A.M. Haeberer, editor, *Proc. of the Seventh International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 389–405. Springer-Verlag, 1999.

4.17 W. Drabent, J. Małuszyński, and P. Pietrzak. Type-based Diagnosis of CLP Programs. *Electronic Notes in Theoretical Computer Science*, 30(4), 2000.

4.18 G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

a specification would correspond to an infinite family of specifications in the present approach.

4.19 T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In G. Kahn, editor, *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 300–309. IEEE Computer Society Press. 1991. (Corrected version available from `http://WWW.pst.informatik.uni-muenchen.de/~fruehwir`)
4.20 J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
4.21 F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, 1997.
4.22 A. Heaton, P. Hill, and A. King. Analysis of Logic Programs with Delay. In *LOPSTR'97, Logic Program Synthesis and Transformation*, volume 1463 of *LNCS*, pages 148–167. Springer-Verlag, 1998.
4.23 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, LNAI, pages 161–192. Springer-Verlag, 1999.
4.24 G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
4.25 J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
4.26 J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.
4.27 P. Mildner. *Type Domains for Abstract Interpretation, A Critical Study*. PhD thesis, Uppsala University, 1999.
4.28 P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
4.29 P. Pietrzak. *Static Incorrectness Diagnosis of CLP(FD)*. Linköping Studies in Science and Technology, Lic. Thesis no. 742, Linköping University, 1998.
4.30 *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science, 1998.
4.31 E. Y. Shapiro. Algorithmic program debugging. In *Proc. Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.

5. Declarative Diagnosis in the CLP Scheme

Alexandre Tessier and Gérard Ferrand

LIFO, BP 6759, F-45067 Orléans Cedex 2, France
email: {Alexandre.Tessier, Gerard.Ferrand}@inria.fr

When a result is computed but it is considered as *incorrect* because it is *not expected*, we consider that we have a *symptom* (of error). The symptom may be a *wrong answer* or a *missing answer*. The role of *diagnosis* is to locate an *error*, that is a limited program fragment responsible for the symptom. The notions of *symptom* and *error* have a meaning only w.r.t. some notion of *expected semantics*. We consider only *declarative semantics*. The user does not need to understand the operational behaviour of the *CLP* system. Symptom and error are connected via some kind of tree and the diagnosis amounts to search for a kind of *minimal symptom* in this tree. Several search strategies are possible. The principles of an implementation are described, with a diagnosis session.

5.1 Introduction

To intuitively introduce the basic notions of *symptom* and *error* let us consider a toy program in the paradigm *clp(FD)*:

```
fac(N,F) :- B#=1, aux(N,B,F).
aux(N,B,P) :- N#=0, B#=P.
aux(N,B,P) :- N#=M+1, C#=N, aux(M,C,P).
```

with an *expected semantics* such as:

– fac is the *factorial function*, that is $fac(N, F) \Leftrightarrow F = N!$
– and $aux(N, B, P) \Leftrightarrow P = N! * B$.

So the last clause should be:

```
aux(N,B,P) :- N#=M+1, C#=N*B, aux(M,C,P).
```

For the *goal:* `N#=<2, fac(N,F)` we have three *computed answer constraints* which are

```
F = 1, N = 0.
F = 1, N = 1.
F = 1, N = 2.
```

A goal is the description of a *relation*. Here the expected relation is: $F = 1, N = 0$ or $F = 1, N = 1$ or $F = 2, N = 2$. So this computed result is considered as *incorrect* because it is *not expected*. Let us consider that we have a *symptom* (of error). But there are two ways to understand that there is a symptom:

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 151–174, 2000.

Firstly we have a *wrong answer*, `F = 1, N = 2.`
To be more formal, a way to express that it is a *wrong answer* is to say that, with respect to the *expected semantics* of the program, the following logic formula (of the form: *computed answer constraint* $\Rightarrow$ *goal*) is *false*:

$$F = 1 \wedge N = 2 \Rightarrow N \leqslant 2 \wedge fac(N, F)$$

It is a first kind of symptom.

Secondly we have a *missing answer*, `F = 2, N = 2.`
Unlike the first kind of symptom, to be able to say that it is a *missing answer* we have to take *all the computed answer constraints* into account: It is missing because among these *computed answer constraints* we do not see `F = 2, N = 2.`
To be more formal, a way to express that it is a *missing answer* is to say that, with respect to the *expected semantics* of the program, the following logic formula is *false*:

$$N \leqslant 2 \wedge fac(N, F) \Rightarrow (F = 1 \wedge N = 0) \vee (F = 1 \wedge N = 1) \vee (F = 1 \wedge N = 2)$$

(since with respect to the *expected semantics*, for $F = 2, N = 2$, the left hand side is true but the right hand side is false). It is a second kind of symptom.

So we have *two* kinds of symptoms: for a *question* (that is a goal) it is possible to have *several* computed *answers* $C_1 \ ; \ \cdots \ ; \ C_i \ ; \ \cdots$ so we may be interested either in a single answer or in the sequence of all the answers. We consider that there are two levels of computation and two kinds of results. At each level the computation is *finite*: At a first level a result is a *single* C_i. If C_i is a *wrong answer* we have a symptom of the first kind. At a second level of computation it is the sequence $C_1 \ ; \ \cdots \ ; \ C_i \ ; \ \cdots$ which is the result. To be more precise, since it is a finite computation, it is a sequence: $C_1 \ ; \ \cdots \ ; \ C_n$ (terminated by *"no more"* answer). *Finite failure* is the particular case where $n = 0$. If an expected answer C is *missing* among $C_1 \ ; \ \cdots \ ; \ C_i \ ; \ \cdots$ we have a symptom of the second kind. With these intuitive motivations we call *positive* the first level, where there is the first kind of symptom, and we call *negative* the second level, where there is the second kind of symptom ([5.8]). It is because we are in a *relational* paradigm that these two levels are shown to be so different.

From an intuitive viewpoint symptoms are "caused" by *errors*. Roughly speaking an error is a limited program fragment responsible for the symptom and the role of *diagnosis* is to locate the error.

For the *positive* level, in our example, the clause

```
aux(N,B,P) :- N#=M+1, C#=N, aux(M,C,P)
```

is erroneous and is responsible for the wrong answer `F = 1, N = 2`, that is to say for the *positive symptom* which is formalised by the (unexpected) formula

$$F = 1 \wedge N = 2 \Rightarrow N \leqslant 2 \wedge fac(N, F)$$

but it is possible to give more information about the "cause" of a symptom: In this small example it is easy to understand that the symptom, that is $fac(2, 1)$ which is false, comes from $aux(2, 1, 1)$, which comes from $aux(1, 2, 1)$, which comes from $aux(0, 1, 1)$. But $aux(0, 1, 1)$ is true whereas $aux(1, 2, 1)$ is false. This transition between true and false is through the erroneous clause *and the constraint* $N = 1, B = 2, P = 1, M = 0, C = 1$, since it is for these values that the body of the erroneous clause is true and its head is false. So the constraint gives more information about the "cause" of the symptom. We consider that it is the *pair* made of this erroneous clause and this constraint which is an *error (incorrectness)*, called *positive error (positive incorrectness)* because it is responsible for the positive symptom.

For the *negative* level the program fragment responsible for the symptom is a "packet of clauses" (all the clauses beginning with a same predicate symbol): the intuitive reason is that some answers are missing because some clause instances are missing. In our example, the "packet"

```
aux(N,B,P) :- N#=0, B#=P.
aux(N,B,P) :- N#=M+1, C#=N, aux(M,C,P).
```

is erroneous and is responsible for the missing answer `F = 2, N = 2.`, that is to say for the *negative symptom* which is formalised by the (unexpected) formula

$$N \leqslant 2 \wedge fac(N, F) \Rightarrow (F = 1 \wedge N = 0) \vee (F = 1 \wedge N = 1) \vee (F = 1 \wedge N = 2)$$

but it is possible to give more information about the "cause" of a negative symptom, like for the positive level. In this small example it is easy to understand that there is again a transition between true and false through the erroneous "packet" *and a constraint*: With respect to the *expected semantics*, $aux(N, B, P)$ is true for $N = 1, B = 2, P = 2$ but it is not possible to have one of the two bodies: either $N = 0, B = P$ or $N = M + 1, C = N, aux(M, C, P)$ true for some values of M, C. A more formal way to express this is to say that for $N = 1, B = 2, P = 2$ the formula

$$\exists M \exists C \ (N = 0 \wedge B = P) \vee (N = M + 1 \wedge C = N \wedge aux(M, C, P))$$

is false w.r.t. the expected semantics. We consider again that it is the *pair* made of the erroneous "packet" and the constraint $N = 1, B = 2, P = 2$ which is an *error (incorrectness)*, called *negative error (negative incorrectness)* because it is responsible for the negative symptom.

The notions of *symptom* and *error* have a meaning only w.r.t. some notion of *expected semantics*. We consider only *declarative semantics* and we presuppose that, during a diagnosis session, the user is able to decide, for some computed answers, if they are *wrong*, or if some expected answer is

missing. In practice a computed answer may be intricate, it is the origin of the *presentation problem* (stressed by Lloyd [5.5]) which may be a difficulty of declarative diagnosis. However this presupposition is necessary to give a meaning to declarative debugging questions. From a conceptual viewpoint the user behaves like an *oracle* which is able to decide if something is wrong or missing (it is the same abstract notion of oracle and the same theoretical framework if we can partially replace the user by a system using some form of specification for the expected semantics of the program).

The notion of positive symptom and positive error comes from Shapiro's seminal work (wrong answer and incorrectness, [5.7]).

For the negative side, the notion of negative symptom and negative error does not correspond to Shapiro's notions (missing answer and insufficiency) which need more complex interaction with the oracle. Theses notions come from W. Drabent, S. Nadjm-Tehrani, J. Małuszyński (*non completely covered atom*, [5.2]).

The rest of the chapter is organised as follows. Section 5.2 defines the theoretical notions of symptom and error. Section 5.3 defines the proof-trees and the diagnosis scheme. Section 5.4 describes the diagnosis algorithms. Section 5.5 links the computation of a result which is a symptom with the proof-tree used in the diagnosis scheme. Section 5.6 describes an implementation. Section 5.7 shows a diagnosis session. Section 5.8 presents a conclusion and future work.

5.2 Basic Notions of Symptom and Error

We use the basic theoretical notions of the *CLP Scheme* ([5.4]). $\mathcal{D}$ is a *constraint domain*. We consider only *definite programs*, that is to say without negation (thanks to disequations and global constraints, negation as failure is less useful than in classical Prolog). A *program* P is supposed to be *normalised* in such a way that only distinct variables are allowed as predicate arguments (all the links between variables are expressed by the constraints). It is a facility to simplify the explanation but it is not a loss of generality of the diagnosis method.

An *interpretation* I for the language of the program is supposed to be given. It is a formalisation of the *expected semantics* of the program. I must be an *expansion* of $\mathcal{D}$, that is to say that it adds to $\mathcal{D}$ an interpretation for the new predicate symbols.

The following definitions are intuitively motivated by the previous introduction.

Definition 5.2.1. $C \rightarrow G$ *(*C *constraint,* G *goal) is a* computed positive symptom *(of* P *w.r.t.* I*) if* C *is a* computed answer *for* G*, but* $C \rightarrow G$ *is* false *in* I*.*

Definition 5.2.2. *A* positive error (positive incorrectness) *(of P w.r.t. I) is a pair made up of a clause $p(\overline{X}) \leftarrow B$ in P and a constraint C such that, for some solution of C, B is true in I but $p(\overline{X})$ is false in I.*

In some logic formulas we use the following *notation*: $\exists_{-A}F$ means quantification over the variables of F which have no occurrence in the expression A.

In the program P a "packet" of clauses

$p(\overline{X}) \leftarrow B_1$
...
$p(\overline{X}) \leftarrow B_m$

is the set of all the clauses of P beginning with a same p. The "packet" is also called the *definition* of the predicate p.

The *completed definition* of the predicate p is the formula

$$p(\overline{X}) \leftrightarrow \exists_{-p(\overline{X})}(B_1 \vee \cdots \vee B_m)$$

It can be rewritten as

$$[p(\overline{X}) \leftarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)] \wedge [p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)]$$

$p(\overline{X}) \leftarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ is merely equivalent to $p(\overline{X}) \leftarrow (B_1 \vee \cdots \vee B_m)$, which is merely equivalent to the "packet" of p.

The other direction: $p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ cannot be simplified so much but it is going to be a *useful notation*. $FI(P)$ (*only-if*(P)) is the set of the formulas

$$p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$$

Remark that we could also mention $IF(P)$, the set of the formulas

$$p(\overline{X}) \leftarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$$

but it is merely equivalent to P. Usually the set of the *completed definitions* is denoted by P^*, it is equivalent to $IF(P) \wedge FI(P)$.

Definition 5.2.3. *$G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ (C_i constraints, G goal) is a* computed negative symptom *(of P w.r.t. I) if for the goal G there exists a* finite SLD-tree *whose* computed answer constraints *are C_1 ; $\cdots$; C_n, but $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$* is false *in I.*

Note that in the particular case $n = 0$ there is a *finite failure* and in such a case $(C_1 \vee \cdots \vee C_n)$ is merely *false* so $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ is $\neg G$.

Definition 5.2.4. *A* negative error (negative incorrectness) *(of P w.r.t. I) is a pair made up of $p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ in $FI(P)$ and a constraint C such that, for some solution of C, $p(\overline{X})$ is true in I but $\exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ is false in I.*

In the rest of this section we explain why, if there is a *computed positive (resp. negative) symptom* then there is a *positive (resp. negative) error*. It is a short and easy verification but this result is purely logical and non constructive. In the next section we set out a refinement of this result giving more information about the connection between symptom and error.

Positive level: At first we recall the basic *soundness* result ([5.4]): If C is a *computed answer constraint* for the goal G then $P \models_{\mathcal{D}} C \rightarrow G$, that is to say $C \rightarrow G$ is true in all the expansions of $\mathcal{D}$ which are models of P. With regard to the question of the applicability to *CLP* systems with *incomplete* solvers, it is interesting to remark that, for this kind of soundness, neither correctness nor completeness is required for the solver (intuitively: if an incomplete solver does not reject C, if C is unsatisfiable then $C \rightarrow G$ is true. If an incorrect solver rejects C, C is not a computed answer, even if it is satisfiable).

Let $C \rightarrow G$ be a computed positive symptom. It is false in I so, thanks to the previous soundness result, I is not a model of P so there is in P some clause $p(\overline{X}) \leftarrow B$ which is false in I, so for some *assignment* a in $\mathcal{D}$, (namely for some values in $\mathcal{D}$), B is true in I but $p(\overline{X})$ is false in I. Such a could give some information about the "cause" of symptoms. To have a positive error it is sufficient to take a C such as a is solution of C. There is always such a C, e.g. *true*. However the more precise C is, the more informative it is.

Negative level: Similar explanation, using another basic *soundness* result: If for the goal G there is a *finite SLD-tree* whose *computed answer constraints* are $C_1 \ ; \ \cdots \ ; \ C_n$, then $FI(P) \models_{\mathcal{D}} G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$.

It is interesting to remark that now, for this kind of soundness, correctness (but not completeness) is required for the solver (intuitively: a satisfiable C_i cannot be removed from the conclusion of the implication, but an unsatisfiable C_i can be added).

5.3 Connection between Symptom and Error via Proof-Trees

Now we consider a notion of symptom which is more general than a computed symptom. Intuitively in these symptoms the constraint may be more informative than in a computed symptom.

Definition 5.3.1. *$C' \wedge C \rightarrow G$ (C, C' constraints, G goal) is a* positive symptom *(of P w.r.t. I) if C is a* computed answer *for G, but $C' \wedge C \rightarrow G$ is* false *in I.*

So if $C' = \mathit{true}$ the symptom is a computed symptom. Likewise:

Definition 5.3.2. *$C' \wedge G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ (C', C_i constraints, G goal) is a* negative symptom *(of P w.r.t. I) if for the goal G there exists a* finite

SLD-tree *whose* computed answer constraints *are* C_1 ; $\cdots$; C_n, *but* $C' \wedge G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ is false *in* I.

At each level of computation, we can get a logical representation of the computation as a *tree*. This tree is a *proof-tree* according to some *rules* as usual in logical formalisms. The rules and the proof-trees are not the same for the two levels of computation but the diagnosis scheme is the same and it is easy to explain it because at each level the diagnosis amounts to search for a kind of *minimal symptom* in this tree.

To clearly explain what are the rules and the proof-trees we use a new toy theoretical example: The program is

$$p(X) \leftarrow q(X), r(X).$$
$$q(X) \leftarrow X > 0.$$
$$q(X) \leftarrow X < 0.$$
$$r(X) \leftarrow X < 1.$$

and the goal is $p(X)$.

At the *positive* level a computation is a *SLD-derivation*. In our example for the goal $p(X)$ there is a *SLD-derivation* giving the computed answer $X > 0 \wedge X < 1$. We can consider this computation as a proof of the formula $X > 0 \wedge X < 1 \rightarrow p(X)$ and to be more precise we can consider the computation as the construction of the following tree:

$$\dfrac{\dfrac{\dfrac{}{X > 0 \rightarrow X > 0}}{X > 0 \rightarrow q(X)} \qquad \dfrac{\dfrac{\dfrac{}{X < 1 \rightarrow X < 1}}{X < 1 \rightarrow r(X)}}{X > 0 \wedge X < 1 \rightarrow X > 0 \wedge r(X)}}{\dfrac{X > 0 \wedge X < 1 \rightarrow q(X) \wedge r(X)}{X > 0 \wedge X < 1 \rightarrow p(X)}}$$

Such a tree is made of *rules* and is called a *proof-tree* according to these rules. There are two kinds of rules:

– For each clause $p(\overline{X}) \leftarrow B$ in P, a "program" rule

$$\frac{C' \rightarrow B}{C \wedge C' \rightarrow C \wedge p(\overline{X})}$$

– and the "logical" rules. In our framework it is convenient to consider *all* the rules

$$\frac{(C_1 \rightarrow G_1), \cdots, (C_n \rightarrow G_n)}{C \rightarrow G}$$

where $C \rightarrow G$ is a *logical consequence* of $(C_1 \rightarrow G_1), \cdots, (C_n \rightarrow G_n)$. These rules are called *logical rules*. For example we can get the previous proof-tree with the two following rules:

$$\frac{C \to G \qquad C' \to C \wedge G'}{C' \to G \wedge G'}$$

$$\frac{}{C \to C}$$

But from the same computation giving the same answer we can also extract another proof-tree:

$$\dfrac{\dfrac{\dfrac{}{X>0 \wedge X<1 \to X>0}}{X>0 \wedge X<1 \to q(X)} \qquad \dfrac{\dfrac{}{X>0 \wedge X<1 \to X<1}}{X>0 \wedge X<1 \to r(X)}}{X>0 \wedge X<1 \to q(X) \wedge r(X)}}{X>0 \wedge X<1 \to p(X)}$$

We can get this second proof-tree with the "program rule" and the following "logical" rules:

$$\frac{C \to G_1 \ \cdots \ C \to G_n}{C \to G_1 \wedge \cdots \wedge G_n}$$

$$\frac{}{C \wedge C_1 \wedge \cdots \wedge C_n \to C}$$

So for *each computed answer constraint* C for a goal G, the formula $C \to G$ is the root of various proof-trees, called *(positive) proof-trees*, according to these various rules, and each of these proof-trees can be easily obtained from the computation namely from the corresponding *SLD-derivation*.

In particular if $C \to G$ is a *computed (positive) symptom* then it is the root of various *(positive) proof-trees*. Let us consider such a proof-tree. Each node of this proof-tree is labelled by a $C' \to G'$. It may be a *symptom node* (the root is a symptom node). Let us consider the notion of *minimal* symptom node where "minimal" is defined w.r.t the binary relation: *x child of y*. So a node is a minimal symptom node if it is a symptom node but no child of it is a symptom node.

To each node of the proof-tree there is a rule which is associated (the label $C \to G$ of the node is the conclusion of the rule). Clearly in a "logical" rule, if the hypotheses are not symptoms then the conclusion is not a symptom. So the rule which is associated to a *minimal* symptom node cannot be a "logical" rule so necessarily it is a "program" rule

$$\frac{C' \to B}{C \wedge C' \to C \wedge p(\overline{X})}$$

Moreover, in this program rule, for some solution of $C \wedge C'$, $p(\overline{X})$ is false in I (since $C \wedge C' \to C \wedge p(\overline{X})$ is a symptom), but B is true in I (since $C' \to B$ is not a symptom), so the clause $p(\overline{X}) \leftarrow B$ and the constraint $C \wedge C'$ of this program rule give a *positive error*.

Clearly there are always minimal symptoms (since proof-trees are finite) and *any way* to find a *minimal symptom* in a *positive proof-tree* gives the localisation of a *positive error*. Moreover, to find a *minimal symptom* the following method is sufficient: To consider only conclusions of program rules (nodes of the form $C \wedge C' \rightarrow C \wedge p(\overline{X})$), and to search for *minimal symptoms with respect to these nodes*. So we have to test formulas $C \wedge C' \rightarrow C \wedge p(\overline{X})$ for symptoms. But such a test amounts to querying (the oracle) whether $C \wedge C' \rightarrow p(\overline{X})$ is *expected*, namely is *true* in I.

At the *negative* level a computation is a *finite SLD-tree*. In our example for the goal $p(X)$ there is a *finite SLD-tree* giving the computed answers $(X > 0 \wedge X < 1)$ and $(X < 0 \wedge X < 1)$. We can consider this computation as a proof of the formula $p(X) \rightarrow (X > 0 \wedge X < 1) \vee (X < 0 \wedge X < 1)$ and to be more precise we can consider the computation as the construction of the following tree:

$$\dfrac{\dfrac{\dfrac{\overline{X>0 \rightarrow X>0} \quad \overline{X<0 \rightarrow X<0}}{q(X) \rightarrow X>0 \vee X<0} \quad \dfrac{\overline{X<1 \rightarrow X<1}}{X>0 \wedge r(X) \rightarrow X>0 \wedge X<1} \quad \dfrac{\overline{X<1 \rightarrow X<1}}{X<0 \wedge r(X) \rightarrow X<0 \wedge X<1}}{q(X) \wedge r(X) \rightarrow (X>0 \wedge X<1) \vee (X<0 \wedge X<1)}}{p(X) \rightarrow (X>0 \wedge X<1) \vee (X<0 \wedge X<1)}$$

Such a tree is made of *rules* and is called a *proof-tree* according to these rules. There are two kinds of rules (with, as usual, appropriate conditions on the free variables, which are not detailed here):

– For each $p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ in $FI(P)$, a "program" rule

$$\dfrac{\text{for } i : B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i}{C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i}$$

– and "logical" rules, for example

$$\dfrac{G \rightarrow \exists_{-G} \bigvee_i C_i \qquad \text{for } i : C_i \wedge G' \rightarrow \exists_{-C_i G'} \bigvee_j C_j^i}{G \wedge G' \rightarrow \exists_{-GG'} \bigvee_i \bigvee_j C_j^i}$$

$$\overline{C \rightarrow C}$$

So if for a goal G there is a *finite SLD-tree* whose *computed answer constraints* are $C_1 \ ; \ \cdots \ ; \ C_n$, then the formula $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ is the root of various proof-trees, called *(negative) proof-trees*, according to these rules, and each of these proof-trees can be easily obtained from the computation namely from the corresponding *SLD-tree* ([5.3], [5.6]).

In particular if $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ is a *computed (negative) symptom* then it is the root of various *(negative) proof-trees*. Let us consider such a proof-tree, in which each node is labelled by a $G' \rightarrow \exists_{-G'}(C'_1 \vee \cdots \vee C'_n)$. A node may be a *symptom node* (the root is a symptom node). Let us consider the notion of *minimal* symptom node where "minimal" is defined w.r.t the

binary relation: *x child of y*. So a node is a minimal symptom node if it is a symptom node but no child of it is a symptom node.

To each node of the proof-tree there is a rule which is associated (the label $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ of the node is the conclusion of the rule). Clearly in a "logical" rule, if the hypotheses are not symptoms then the conclusion is not a symptom. So the rule which is associated to a *minimal* symptom node cannot be a "logical" rule so necessarily it is a "program" rule

$$\frac{\text{for } i : B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i}{C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i}$$

Moreover in this program rule, for some solution of C, since $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is a symptom, $p(\overline{X})$ is true in I but $\exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is false in I. But the $B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i$ are not symptoms. Let us suppose $\exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ is true in I, then, for some i, $\exists_{-\overline{X}} B_i$ is true, so $\exists_{-B_i} \bigvee_j C_j^i$ is true, so $\exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is true which is a contradiction. So $\exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ is false in I. So $p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \cdots \vee B_m)$ and the constraint C of this program rule give a *negative error*.

Clearly there are always minimal symptoms (since proof-trees are finite) and *any way* to find a *minimal symptom* in a *negative proof-tree* gives the localisation of a *negative error*. Moreover, to find a *minimal symptom* the following method is sufficient: To consider only conclusions of program rules (nodes of the form $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$), and to search for *minimal symptoms with respect to theses nodes*. Such a minimal symptom can be found by testing if some $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ are symptoms. But this test amounts to querying (the oracle) whether $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C_j^i$ is *expected*, namely is *true* in I.

5.4 Diagnosis Algorithm

So for each level of computation we have a notion of proof-tree. And any way to find a *minimal symptom* in a proof-tree gives the localisation of an *error*. Let us consider a proof-tree rooted by a (computed) symptom. The diagnoser queries the oracle about the labels of the nodes of the proof-tree, it is a test: "is this a symptom?". The oracle does not need to understand the operational behaviour of the system.

The objective of the diagnoser is to locate a minimal symptom (a minimal symptom node). With this end in view the diagnoser uses a strategy in order to choose the node which corresponds to the next query to the oracle.

Let us assume that each node of the proof-tree can be either *expected* (it is not a symptom) or *unexpected* (it is a symptom) or *unknown* (it is not yet determined). The diagnosis algorithm is the following:

while no minimal symptom appears in the proof tree
  choose an *unknown* node according to some strategy
  query the oracle about the chosen node in order to determine if it is *expected* or *unexpected*
show the error associated with a minimal symptom

It is easy to show that this algorithm is correct and that it is complete in the sense that it always founds an error (the proof-tree is finite).

In general the oracle is the user and we have to take into account the fact that the user cannot answer some queries of the diagnoser. Thus we have a fourth category of nodes: the *dontask* nodes (the oracle cannot determine if it is a symptom). It is trivial to see that this possibility involves that the diagnoser is not always able to locate a minimal symptom. Moreover it involves that the strategies used to choose a node are more intricate and sometimes cannot choose a node.

As an example, we describe here the strategies *Top-Down* and *Divide&Query* which are the most useful in practice.

The Top-Down strategy follows a path of *unexpected* nodes from the root until it reaches an *unknown* node. This *unknown* node is the chosen node. Because of the *dontask* nodes, this strategy does not always find a node to query (see Fig 5.7). It looks like a prefix traversal of the proof-tree where sub-proof-trees rooted by an *expected* or *dontask* node are removed (see Fig. 5.1).

The relation $topdown(Node, ChosenNode)$ provides the node chosen ($ChosenNode$) by the Top-Down strategy in the tree rooted by $Node$. If the strategy cannot conclude $ChosenNode = notfound$.

$$\frac{topdown(Node, ChosenNode)}{\begin{array}{l}\text{if } Node \text{ is } unknown\\ \quad \text{then } Node = ChosenNode\\ \quad \text{else if } Node \text{ is } unexpected\\ \quad\quad \text{then } topdown'(children(Node), ChosenNode)\\ \quad\quad \text{else } ChosenNode = notfound\end{array}}$$

$$\frac{topdown'([], notfound)}{\frac{topdown'([Node|Children], ChosenNode)}{\begin{array}{l}topdown(Node, ChosenNode')\\ \text{if } ChosenNode' = notfound\\ \quad \text{then } topdown'(Children, ChosenNode)\\ \quad \text{else } ChosenNode = ChosenNode'\end{array}}}$$

Fig. 5.1 shows an example of proof-tree and the node chosen by the Top-Down strategy. The proof-tree is drawn as usual: the root of the proof-tree is at the top of the drawing, the parent relation of the proof-tree links a node which is the conclusion of a rule with its hypotheses which are its children on the drawing. In the drawing the following notation are used: *unexpected*

nodes are labelled by "N", *expected* nodes by "Y", *dontask* nodes by "X" and *unknown* nodes by "?").

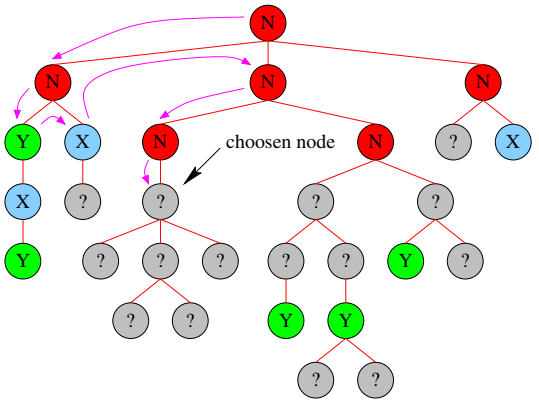


Fig. 5.1. Top-Down Strategy

The basic principle of the Divide&Query strategy is to choose a node in order to divide the search space in two parts:

- if the node is *expected*, the subtree rooted by this node can be removed from the search space,
- if the node is *unexpected*, each node which is not a descendant of this node can be removed from the search space.

The point is to choose a node such that there is as much *unknown* nodes in its subtree as out of its subtree (considering that all subtrees rooted by an *expected* node are removed from the search space). Note that it is not always possible to have as much *unknown* nodes in the subtree as out of it, but the strategy chooses the nodes which is the most near this condition.

The Divide&Query strategy here is an improvement of the Divide-and-query strategy of [5.7]. It really chooses the best node (that is the node which better divides the search space in two parts) and it takes into account *dontask* nodes and the fact that the strategy may be changed during a diagnosis session.

The tree considered is the tree rooted by an *unexpected* node with the least number of *unknown* and *dontask* nodes, but where a minimal symptom is always possible (remember the *dontask* nodes).

In Fig. 5.2 we use the same notations as in the previous example. In addition, the integer at the left of the nodes is the number of *unknown* or *dontask* nodes in their subtrees. The 1^{st} subtree and the 4^{th} subtree are not considered because of the *dontask* nodes: if the *unknown* nodes are *expected* no minimal symptom can be detected. The 3^{rd} subtree is considered because the number of *unknown* nodes is lower than in the 2^{nd} subtree. The 3^{rd} subtree has 5 *unknown* nodes so the Divide&Query strategy chooses the root of the

subtree with 3 *unknown* nodes: when the user answer the query the subtree will have 2 *unknown* nodes inside and 2 *unknown* nodes outside $((5-1)/2)$.

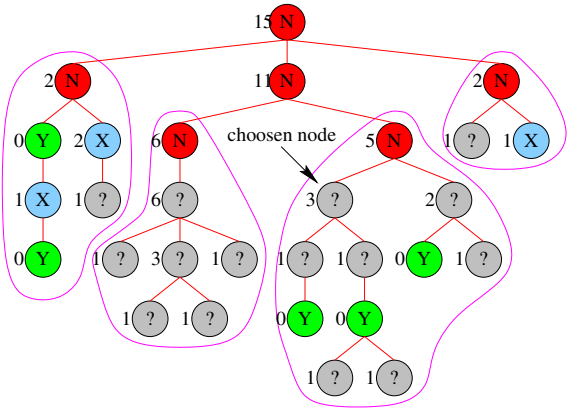


Fig. 5.2. Divide&Query Strategy

It is possible to build pathological cases (where whatever node you choose there is few nodes in one side and a lot of nodes in the other side) but in practice this strategy queries about $\log_2(N)$ nodes (where N is the number of *unknown* nodes). For instance, in a proof-tree with 1,000,000 nodes there is less than 20 queries (try to find an error without declarative diagnosis!).

5.5 Abstract Proof-Trees

As said previously, only the nodes which corresponds to conclusion of program rules can be minimal symptom nodes. So, in order to decrease the size (number of nodes) of proof-trees, we define *abstract proof-trees*. Let us assume that a proof tree is given, the abstract proof tree which corresponds to the proof tree is defined as follow:

1. the root of the abstract proof-tree is the root of the proof-tree,
2. y is a child of a node x in the abstract proof-tree if
   - y is the conclusion of a program rule in the proof tree,
   - y is a descendant of x in the proof-tree,
   - and there is no other conclusion of program rules between x and y in the proof-tree.

The abstract proof trees A which corresponds to a proof tree B can be also defined as an abstraction of the proof tree B in the sense of Chapter 8: all the conclusion of program rules are selected (in addition of the root).

In the following we say proof-tree instead of abstract proof-tree.

The user invokes the diagnoser when a (positive or negative) symptom appears at the end of a (positive or negative) computation. The computation is either a SLD-derivation (a branch of a SLD-tree) or a SLD-tree. But the diagnoser uses a proof-tree!

The point is that it is possible to compute directly a (positive or negative) proof-tree rooted by the (positive or negative) computed symptom from a (positive or negative) computation, that is from (a branch of) a search-tree.

In order to simplify, the computation rule is assumed to be without co-routining (for instance the standard computation rule of Prolog), then proof-trees can be deduced from search-tree using the notion of *erasing* defined below. [5.6] shows an extension to any computation rule.

The main useful notion is: let x and y be two nodes of the search-tree, y is a node where x is *erased* if

- A_i is the atom selected at the node x in its goal (the goal on x is $A_1, \ldots, A_i, \ldots, A_n$ and the store is C),
- y is a descendants of x where A_i is fully solved (the goal on y is $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$ and the store is $C \wedge C'$ where C' is a computed answer to A_i).

Let x be a node of the search-tree, we denote by

- $store(x)$ the set of constraints accumulated from the root until the node x,
- $select(x)$ the atom selected at the node x in its goal,
- $erased(x)$ the set of nodes where x is erased,
- $stchildren(x)$ the set of children of x in the search-tree.

Let C be a computed answer constraint for the goal G such that $C \rightarrow G$ is a positive symptom. C is made of the constraints accumulated along a success branch of the search-tree. The set of nodes of this branch is denoted by *branch*.

In order to define the positive proof tree which corresponds to the positive symptom computed, we have to determine:

1. the root of the proof-tree;
2. the binary relation: x *child of* y in the proof-tree;
3. the formula (query) which labels a node of the proof-tree.

We consider, in order to simplify, that the nodes of the proof-tree are a subset of the nodes of the search-tree, but the labels of a node is different depending on whether it is considered as a node of the proof-tree or a node of the search-tree.

First, the root of the positive proof-tree is the root of the search-tree.

Secondly, the list L of children of a node x of the positive proof-tree is given by the relation $+children$:

$+children(x, L)$
$\{y\} = stchildren(x) \cap branch$
if $y \in erased(x)$
then $L = []$
else $+ children'(x, y, L1),\ L = [y|L1]$

$+children'(x, y, L)$
$\{z\} = erased(y) \cap branch$
if $z \in erased(x)$
then $L = []$
else $+ children'(x, z, L1),\ L = [z|L1]$

Finally, the formula associated with a node x of the positive proof-tree is $store(y) \rightarrow select(x)$ where y is the success leaf of $branch$. We use the constraint $store(y)$ in the formula because it is the most precise we can know, but as said in Section 5.3 there exists various proof-trees.

For example, let us consider the small program (without constraints and variables to be more concise, in other words the constraints are always $true$):

$p \leftarrow q, r.$
$p \leftarrow q, a.$
$r \leftarrow a.$
$q \leftarrow w.$
$w.$
$a \leftarrow z.$
$a.$

The positive proof-tree which corresponds to the first answer to the goal p is given by Fig. 5.3. The nodes of the search-tree used by the proof-tree have been duplicated for the legibility of the drawing. The selected atom in the goals is underlined in the search-tree. For example, p has two children in the positive proof-tree: its child q in the search-tree, the node r where q is erased. Note that the node $\square$ where r is erased is also the node where p is erased.

On Fig. 5.3 you can recognise a more classical notion of proof-tree in logic programming (a node corresponds to the head of a clause of the program and its children correspond to the body of the clause).

Let $C_1, \ldots, C_n$ be the computed answer constraints for the goal G such that $G \rightarrow \exists_{-G}(C_1 \vee \cdots \vee C_n)$ is a negative symptom.

In order to define the negative proof tree which corresponds to the negative symptom computed, we have to determine:

1. the root of the proof-tree;
2. the binary relation: x *child of* y in the proof-tree;
3. the formula (query) which labels a node of the proof-tree.

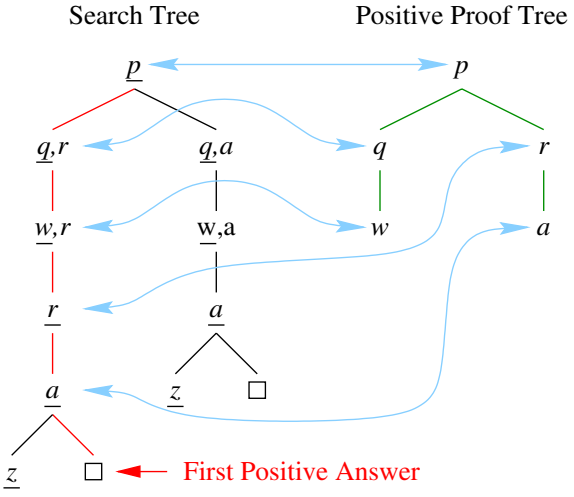


Fig. 5.3. From a branch of the search-tree to the positive proof-tree.

First, the root of the negative proof-tree is the root of the search-tree.

Secondly, the list L of children of a node x of the negative proof-tree is given by the relation $-children$:

$$\frac{-children(x, L)}{\begin{array}{|l} \text{let } S \text{ be the list of nodes of } (stchildren(x) \setminus erased(x)) \\ -children'(x, S, L) \end{array}}$$

$$\frac{-children'(x, [], [])}{-children'(x, [y|S], [y|L])}$$
$$\begin{array}{|l} -children''(x, y, L1) \\ -children'(x, S, L2) \\ append(L1, L2, L) \end{array}$$

$$\frac{-children''(x, y, L)}{\begin{array}{|l} \text{let } S \text{ be the list of nodes of } (erased(y) \setminus erased(x)) \\ -children'(x, S, L) \end{array}}$$

Finally, the formula associated with a node x of the negative proof-tree is $store(x) \wedge select(x) \rightarrow \bigvee_{y \in erased(x)} store(y)$.

For example, let us consider again the previous program:

$p \leftarrow q, r.$
$p \leftarrow q, a.$
$r \leftarrow a.$
$q \leftarrow w.$
$w.$
$a \leftarrow z.$
$a.$

The negative proof-tree which corresponds to the search-tree for the goal p is given by Fig. 5.4. The nodes of the search-tree used by the proof-tree have been duplicated for the legibility of the drawing. The selected atom in the goals is underlined in the search-tree. For example, p has four children in the negative proof-tree: its children q and q in the search-tree, the node r where the first q is erased, the node a where the second q is erased. Note that the nodes where r and a are erased are also the nodes where p is erased.

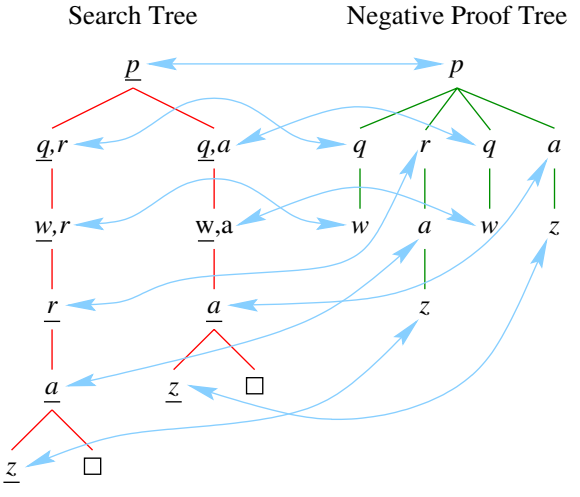


Fig. 5.4. From the search-tree to the negative proof-tree.

5.6 Implementation

The positive part of the declarative diagnoser has been implemented and tested on the INRIA platform: TkCalypso, developed in the DiSCiPl project. At the time of writing this book, implementation of the negative part is in progress.

TkCalypso is an extension of GNU-Prolog [5.1]. It includes a graphical interface (Fig. 5.5) and some debugger modules. Each module can be plugged or unplugged. Fig. 5.6 shows the structure of TkCalypso with the three modules that are actually implemented: search-tree visualisation, static debugger and declarative diagnoser. Communications between GNU-Prolog and the modules and the graphical interface are handled by the "Core/Gestionnary" packages. This section describes the main features of the module called "Declarative Diagnoser".

When a goal is given to TkCalypso, it stores informations on the search-tree in order to recompute it efficiently and make post-mortem analysis of the search-tree.

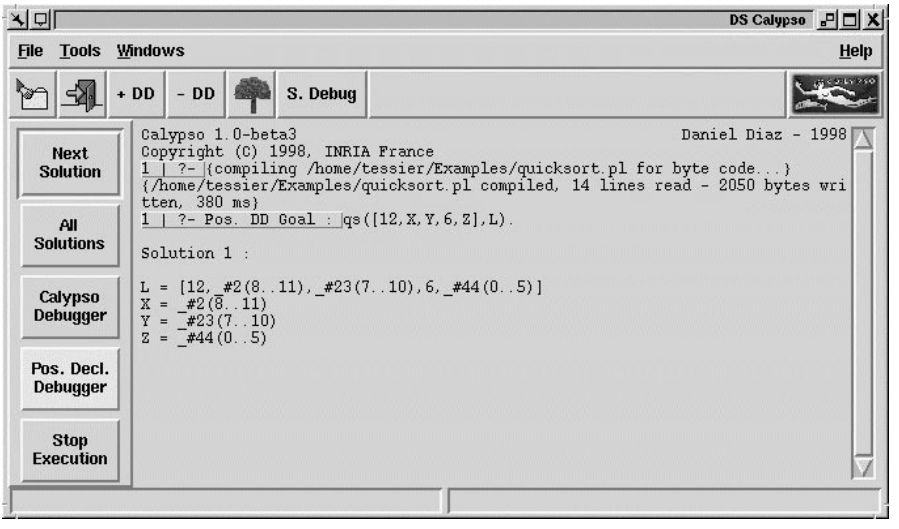


Fig. 5.5. Graphical interface: a positive symptom

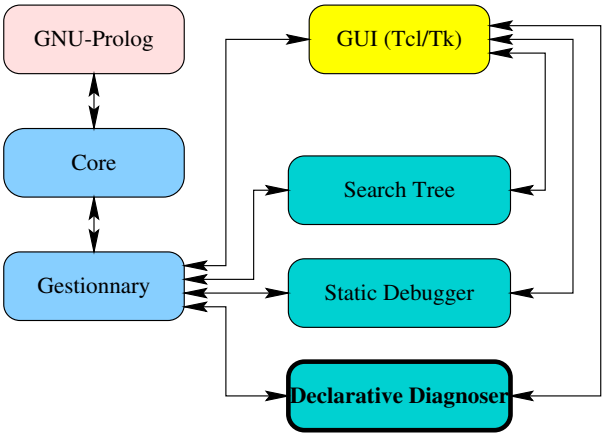


Fig. 5.6. Architecture of the TkCalypso platform.

If the user notices a computed symptom, then the declarative diagnoser is called and the parent relation of the (positive or negative) proof-tree is computed dynamically from the search-tree.

Once we have a (positive or negative) proof-tree, the diagnosis principle is always the same: choose a node of the proof-tree (according to some strategy), check if it is expected or not expected, until a minimal symptom is founded.

Several strategies have been implemented in order to choose the node to query: Top-Down, Bottom-Up, Divide&Query, Nearby-Error and User-Guided (Top-Down and Divide&Query are described in Section 5.4).

Built-in predicates are known as correct predicates, but also user predicates could be known as correct. For example, when the static analysis (see Chapter 2) has proved their correctness. Another example is when the user is convinced that some predicates are correct. So the user can set a list of correct predicates, which will be used by the diagnoser.

It is possible that the user does not want to be questioned on some predicates. For example, the semantics of the predicate is very intricate and the user want to suspend the queries on that predicate as long as possible. So the user can set a list of predicates which must not be questioned.

The status of a node of the proof-tree is more precise that the ones describe in Section 5.4, because we want to know the origin of this status, it can be:

- *unknown* when the predicate associated to the node is not a built-in and the node has not been queried,
- *expected*(*user*) when the user said that the node is expected,
- *expected*(*list*) when the predicates associated to the node is in the list of correct predicates (when the user is convinced that the predicate is correct),
- *expected*(*system*) when the predicate associated to the node is a built-in predicate (the are not suspected!),
- *unexpected*(*user*) when the user said that the node is not expected,
- *dontask*(*user*) when the user does not want to answer the query associated to the node (for example the query is very intricate), note that the user can come back later to the query associated with this node,
- *dontask*(*list*) when the predicates associated to the node is in the list of predicates that the user does not want to answer (for example the user does not know the semantics of the predicates or the user wants to delay the queries about the predicate).

The user has the possibility to dynamically add some predicates to the list of correct predicates or remove some predicates from it. If the user adds a predicate, each *unknown* or *dontask*(_) node concerning the predicate is labelled by *expected*(*list*); if a node is *unexpected*(*user*) then the problem is given to the user: either remove the predicate from the correct predicate list or label the node as *expected*(*user*). When the user removes a predicate of the list, each *expected*(*list*) node concerning the predicate are labelled by *unknown*.

The user can also change dynamically the list of predicates which must not be questioned. If the user adds a predicate to the list, each *unknown* node concerning the predicate becomes a *dontask*(*list*) node. If the user removes a predicate of the list, each *dontask*(*list*) node concerning the predicate becomes an *unknown* node.

The graphical interface (see Fig.5.7) of the diagnoser works like an hypertext navigator: the user can navigate between queries, because its answers are seen as hyper-link between queries (it is possible to go back, go forward, see

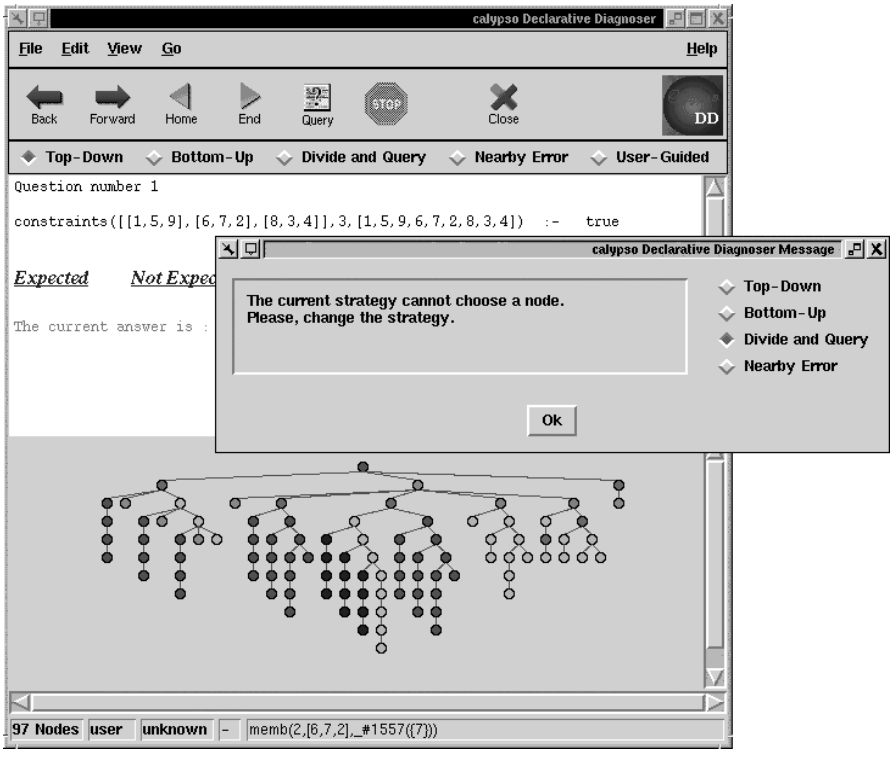


Fig. 5.7. Declarative Diagnoser Interface

an history...). The proof-tree is displayed with informations about the nodes. For example, the user can see the status of a node

- with colours: red = *unexpected*, grey = *unknown*, blue = *dontask*, green = *expected*,
- and with shades: light = *system*, medium = *user*, dark = *list*.

The user can also choose the query with the mouse on the proof tree: it is the "User Guided" strategy.

The query $C \rightarrow A$ is displayed as A :- $\exists\__A C$. The diagnoser uses a trivial store simplification in order to simplify the constraint $\exists\__A C$ in the query. But this point has to be improved (this is discussed later in Section 5.8).

5.7 A Diagnosis Session

Let us consider the following QuickSort program which is intended to sort a list of finite domain terms:

```
qs([], []).
qs([Pivot | List], SortList) :-
        partition(Pivot, List, MinList, MaxList),
        qs(MinList, SortMinList),
        qs(MaxList, SortMaxList),
        append([Pivot|SortMinList], SortMaxList, SortList).

partition(_, [], [], []).
partition(Pivot, [X | List], [X | MinList], MaxList) :-
        X #< Pivot,
        partition(Pivot, List, MinList, MaxList).
partition(Pivot, [X | List], MinList, [X | MaxList]) :-
        X #> Pivot,
        partition(Pivot, List, MinList, MaxList).
```

In the expected model of `qs(A,B) A` is a list of distinct finite domain terms, `B` is a permutation of `A` and `B` is an increasing list (for example $X < 5 \rightarrow qs([5, X, 7], [X, 5, 7])$ is expected). In the expected model of `partition(A, B, C, D) A` is a finite domain term, `B` is a list of finite domain terms, `C` is the list of members of `B` which are lower than `A`, `D` is the list of members of `B` which are greater than `A`, `B` is obtained by a fusion of `C` and `D`.

Fig. 5.5 shows that for the goal `qs([12,X,Y,6,Z],L)` the first answer is:

```
L = [12, _#2(8..11), _#23(7..10), 6, _#44(0..5)]
X = _#2(8..11)
Y = _#23(7..10)
Z = _#44(0..5)
```

(`X = _#2(8..11)` means that `X` is a finite domain variable whose domain is 8..11). It is a positive symptom: `L` is not an increasing list. So we call the positive declarative diagnoser module of TkCalypso and a new window appears on the screen, with a drawing of the positive proof tree.

The diagnoser queries the user on some nodes of the proof-tree. For example in Fig. 5.8, the user will answer (click on) "*Not Expected*" because the last constraint in the store is `_#44(0..5) #< 6`, so `[6,_#44(0..5)]` is not an increasing list. The store simplification has been disabled on the figure and we see that without simplification the store quickly becomes unreadable.

After several queries the diagnoser finds a minimal symptom and shows the corresponding error. Fig 5.9 shows an error, first it displays the incorrect clause and next the error: a clause instance and a constraint store. The problem is that in `append([A|F],G,C)` the pivot `A` should be between the list `F` and the list `G`: `append(F,[A|G],C)`.

Sometimes it is not easy to fix the error provided by the declarative diagnoser, but the user is sure that the clause is incorrect, so the user does not need to search elsewhere in the program. Thus declarative diagnosis is very efficient especially for large programs (with a lot of clauses) or for large

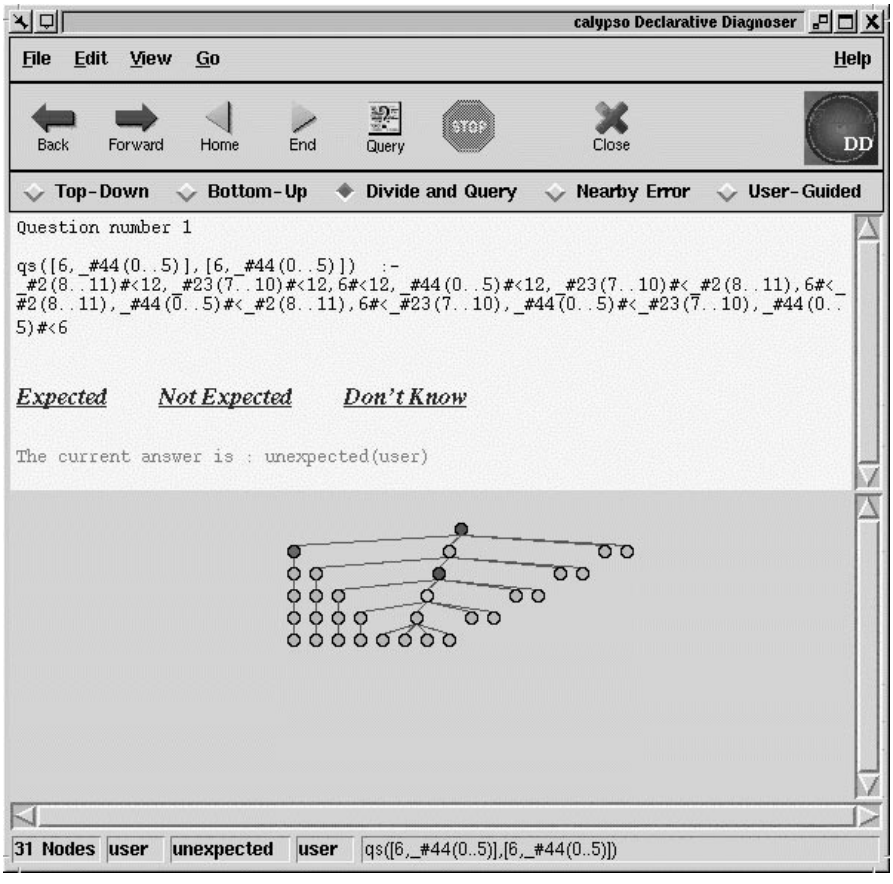


Fig. 5.8. A query

computations (search-tree with a lot of nodes). It is also a good tool for educational purpose.

5.8 Conclusion

The main remaining task concerns the interaction with the oracle.

Of course, answers to previous queries may be used by the declarative diagnoser in order to automatically answer to some other queries: Let us consider a query "$C \to A$ expected?" If it is stored that $C' \to A$ is expected and if $C \to C'$ is true then $C \to A$ is expected. If it is stored that $C' \to A$ is unexpected and if $C' \to C$ is true then $C \to A$ is unexpected. Likewise for the negative side.

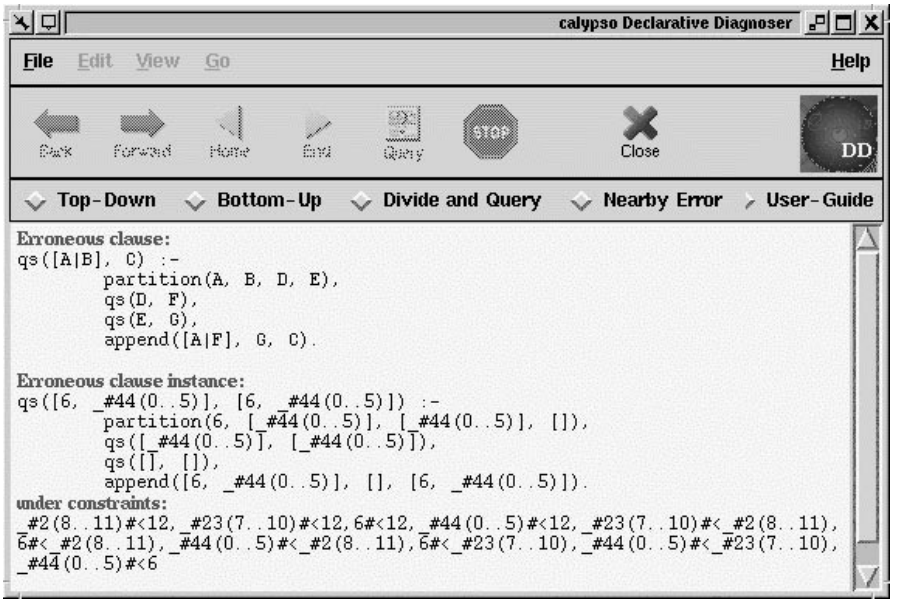


Fig. 5.9. An error

The difficulty is to decide if $C \to C'$ is true (or in general $C_1 \vee \cdots \vee C_m \to C'_1 \vee \cdots \vee C'_n$), that is the *entailment problem*.

It is interesting to study how assertions defined in Chapter 1 could be used to answer to queries of the declarative diagnoser. Then some assertions are viewed as a partial specification of the expected semantics of the program.

Despite these techniques it is not possible to completely avoid interaction with the user. So, works in progress concerns the *presentation problem*, that is to show queries in an understandable form. Variable elimination, redundant constraint elimination, constraint simplification and approximation may be useful methods to present queries to the user.

References

5.1 D. Diaz. A Native Prolog Compiler with Constraint Solving over Finite Domains Edition 1.0, for GNU Prolog version 1.0.0, 1999. `http://www.gnu.org/software/prolog/`

5.2 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. The MIT Press, 1989.

5.3 G. Ferrand and A. Tessier. Clarification of the bases of Declarative Diagnosers for CLP. Deliverable D.WP2.1.M1.1-1. Debugging Systems for Constraint Programming (ESPRIT 22532), 1997. `http://discipl.inria.fr/`

5.4 J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
5.5 J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.
5.6 B. Malfon and A. Tessier. An Adaptation of Negative Declarative Error Diagnosis to any Computation Rule. Deliverable D.WP2.1.M2.1-1. Debugging Systems for Constraint Programming (ESPRIT 22532), 1998. `http://discipl.inria.fr/`
5.7 E. Y. Shapiro. Algorithmic Program Debugging. ACM Distinguished Dissertation. The MIT Press, 1982.
5.8 A. Tessier. Correctness and Completeness of CLP Semantics revisited with (Co-)Induction. Deliverable D.WP2.1.M2.1-2. Debugging Systems for Constraint Programming (ESPRIT 22532), 1998. `http://discipl.inria.fr/`

6. Visual Tools to Debug Prolog IV Programs

Pascal Bouvier

Société PrologIA, Parc Technologique de Luminy, Case 919,
F-13288 Marseille Cedex 9, France
email: bouvier@prologianet.univ-mrs.fr

Like any non-deterministic language, Prolog IV is difficult to use and large Prolog IV programs are quite challenging to debug —not to mention the fact that Prolog IV has a rich and complex constraint mechanism ... Although debugging tools already exist in previous versions of the Prolog IV system, their paradigm, as local debuggers/viewers, doesn't give the programmer a good picture of the whole execution process. This chapter presents the Prolog IV Visual Debugger and explains how it is used with the other tools for program debugging and profiling.

6.1 Introduction

The Prolog IV [1] Development System has a debugger to monitor the details of execution of a user's program. This debugger is based upon the so called Prolog IV Box Model Debugger which belongs to the Prolog IV core and dispatches information to various viewers. Prior to version 1.2 of Prolog IV, the user-visible part of the debugger was composed of the two following execution viewers, running synchronously:

- the box model viewer, communicating with the user by means of a textual "terminal".
- the source level viewer which displays and highlights the current source code and facilitates the choice of which currently living variables are to be printed.

Prolog IV version 1.2 incorporates a new viewer, the Execution Tree Viewer, working synchronously with the two others. Fig. 6.1 shows how the debugging system is organised.

After a short description of the core debugger, some explanations about the execution trees are given (Section 6.2), followed by more practical discussions about the use of these new tools (Sections 6.3 and 6.4). The chapter ends with a few words about implementation and a conclusion.

[1] Prolog IV is a PrologIA company product, and has benefited from work in ESPRIT projects PRINCE 5246 and ACCLAIM 7195.

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 177–190, 2000.

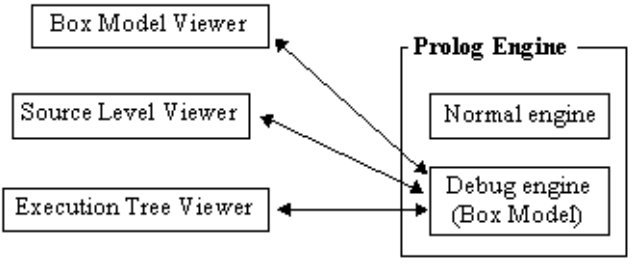


Fig. 6.1. The Prolog IV debugging system

6.2 Presentation of the Execution Tree Viewer

The main tool for graphic visualisation of a running Prolog IV program is the Execution Tree Viewer, whose purposes can be summarised as follows:

- to give a full view of an execution-tree (the shape of the tree gives precious indications about deterministic and combinatorial parts of the execution, the distribution of solutions, etc),
- to present a view of a proof (*i.e.* any branch from the top node to a given node, whatever it is a success or a failure node),
- to allow rapid access to a given point in the execution-tree,
- to help restart the execution up to a given point.

Before defining execution-trees, let us introduce the Prolog IV Box Model, since this is the core of the Prolog IV debugger, and, as such, the place where debugging tools as viewers have to be connected to, to receive and send information.

6.2.1 The Box Model

The Prolog IV debugger is based on the "box model", commonly found in several prolog systems. The key concept of this model is that *every call is considered as a box*; a glass-box or a black-box depending on whether or not the user needs detailed traces. Boxes in this model have four ports. Two more ports are available in Prolog IV, enabling the user to peek inside the boxes. These ports are related to precise points of execution of the Prolog IV engine, when it executes a goal, namely:

- the activation of a rule block (`CALL`),
- the termination of one of the block's rules (`EXIT`),
- a backtracking inside the block (`REDO`),
- the deactivation of the block, because all rules have been tried (`FAIL`),
- the selection of the n-th rule of a block (`RULE`),
- the successful unification of the rule head with the current call (`OK`),

The last two ports are secondary ones, inside the box.

The behaviour of a box is naturally non-deterministic, and the same box may be solicited several times in order to give several different answers. Sequentially, a box is entered via a `CALL` port. It is then exited via the `EXIT` port when it succeeds (a solution has been found). When choices have been left pending, the box is re-entered through the `REDO` port. It may be exited again through the `EXIT` port if a solution is found. When there are no more solutions available, the `FAIL` port is used to definitively exit the box.

The display of ports in the Box Model Viewer respects the same template, as in the following example:

```
4 [2] EXIT (r1) : dessert(fruit)
```

Here is the description of the various fields used:

`4` : Identification of the call: This is the running number of the box from which the call has been performed.

`[2]` : Depth of the call: This is the nesting level of the box (*i.e.* the number of calling (and nested) boxes around the current box). This number is of course the same for all the calls found in the same rule tail (or query).

`EXIT` : This is the name of the port, *i.e.* one of the strings `CALL`, `EXIT`, `REDO`, `FAIL`, `RULE` or `OK`.

`(r1)` : The number of the rule applied. When followed by a star, it shows that this rule is the last in the block.

`dessert(fruit)` : The literal currently being handled. It is displayed in its current instantiation state.

The ports are displayed just before they are traversed.

6.2.2 Execution Trees

When a prolog program runs, the prolog engine conceptually draws an execution-tree. In this implementation we choose to display execution-trees defined[2] as follows:

- the root of the tree is at the top, it is the first call of the query;
- conjunctions (call-nodes) correspond to direct calls and continuations; they are connected vertically;
- alternatives (choice-nodes) for a given call represent each an attempt to match the call against the head of a rule; choice-nodes for a given call are connected horizontally.

Note that only successful matches are displayed if the option "Show all tries" is not set. See section 6.3.4 for more information about this option.

Here are our conventions to represent nodes graphically:

[2] This is related to classical AND-OR trees definitions, with simplifications concerning AND-nodes.

– a call-node is shown as a big coloured square;
– a choice-node is shown as a small white square.

An execution of a prolog program corresponds to the pre-order traversal of the execution-tree. We now introduce several small programs and their execution-trees to help figure out the general shapes of these trees. This first example serves to show how alternatives are displayed.

```
alt(a).
alt(b).
alt(c).
alt(d).
```

When the query `alt(X)` is launched, the log as printed by the Box Model Viewer follows, and the execution-tree obtained is depicted in Fig. 6.2. This tree contains four branches, *i.e.* four ways to go from top to bottom; moreover, all branches are successful.

```
    1(1)[1]CALL    : alt(_259)
     2(1)[2]RULE(r1): alt/1(_259)
     2(1)[2]OK(r1): alt/1(a)
     2(1)[2]EXIT(r1): alt/1(a)
X = a
  1(0)[0]REDO(r1): alt/1(_259)
    2(1)[2]RULE(r2): alt/1(_259)
    2(1)[2]OK(r2): alt/1(b)
    2(1)[2]EXIT(r2): alt/1(b)
X = b
  1(0)[0]REDO(r2): alt/1(_259)
    2(1)[2]RULE(r3): alt/1(_259)
    2(1)[2]OK(r3): alt/1(c)
    2(1)[2]EXIT(r3): alt/1(c)
X = c
  1(0)[0]REDO(r3): alt/1(_259)
    2(1)[2]RULE(r4*): alt/1(_259)
    2(1)[2]OK(r4*): alt/1(d)
    2(1)[2]EXIT(r4*): alt/1(d)
X = d
    2(1)[2]FAIL(r4*): alt/1(_259)
```

The second example is used to show (sub)calls and how they are chained. The program is deterministic (there are no alternatives):

```
a :- b, c, d.
b :- e.
c :- f, g.
d :- h.
e. f. g. h.
```

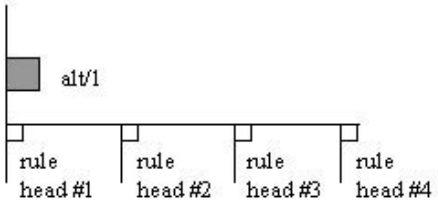


Fig. 6.2. Example of an execution-tree (demonstrating choice-nodes)

Given a query `a`, we obtain the execution-tree presented in Fig. 6.3 containing one (successful) branch. The execution as printed by the Box Model Viewer follows (`OK` ports have been omitted, for the sake of simplicity):

```
      1(1)[1]CALL    : a
       2(1)[2]RULE(r1*): a/0
       2(2)[2]CALL    : b
        3(2)[3]RULE(r1*): b/0
        3(3)[3]CALL    : e
         4(3)[4]RULE(r1*): e/0
         4(3)[4]EXIT(r1*): e/0
        3(2)[3]EXIT(r1*): b/0
       2(4)[2]CALL    : c
        5(4)[3]RULE(r1*): c/0
        5(5)[3]CALL    : f
         6(5)[4]RULE(r1*): f/0
         6(5)[4]EXIT(r1*): f/0
        5(6)[3]CALL    : g
         7(6)[4]RULE(r1*): g/0
         7(6)[4]EXIT(r1*): g/0
        5(4)[3]EXIT(r1*): c/0
       2(7)[2]CALL    : d
        8(7)[3]RULE(r1*): d/0
        8(8)[3]CALL    : h
         9(8)[4]RULE(r1*): h/0
         9(8)[4]EXIT(r1*): h/0
        8(7)[3]EXIT(r1*): d/0
       2(1)[2]EXIT(r1*): a/0
true
         9(8)[4]FAIL(r1*): h/0
        8(7)[3]FAIL(r1*): d/0
         7(6)[4]FAIL(r1*): g/0
         6(5)[4]FAIL(r1*): f/0
        5(4)[3]FAIL(r1*): c/0
         4(3)[4]FAIL(r1*): e/0
        3(2)[3]FAIL(r1*): b/0
       2(1)[2]FAIL(r1*): a/0
```

The right hand part of Fig. 6.3 represents the (indented) call cascade, deduced from the rules' body. This indentation shows the various boxes' levels. Note that this rule structure is not reflected in the execution-tree itself (left hand part of figure). Execution-trees as defined in this chapter only show how calls are linked together with respect to time.

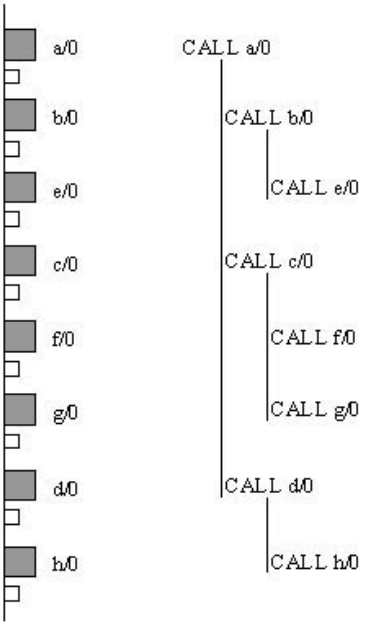


Fig. 6.3. Example of an execution-tree (demonstrating call-nodes)

To have a clearer conception of execution-trees, let us now introduce a prolog program whose execution-tree contains both types of node:

```
meal(H,M,D) :-
    horsDoeuvre(H),
    mainCourse(M),
    dessert(D).

horsDoeuvre(radishes).
horsDoeuvre(pate).

mainCourse(M) :- meat(M).
mainCourse(M) :- fish(M).

meat(beef).
meat(pork).
```

```
fish(sole).

dessert(fruit).
```

When launched with query `meal(H,M,D)`, the execution-tree is as shown in Fig. 6.4. The tree is made up of six (successful) branches.

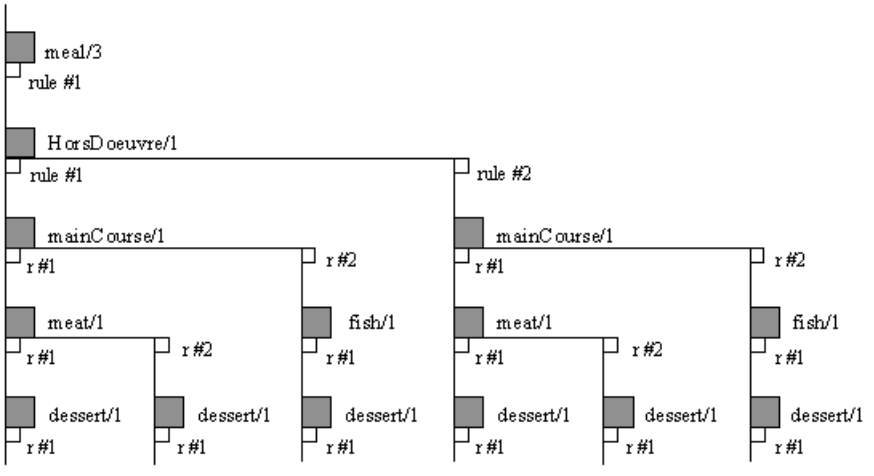


Fig. 6.4. Execution-tree of `meal/3` (flat view)

6.2.3 The Prolog IV Execution Tree Viewer

Fig. 6.5 depicts a typical execution-tree in a canvas of the Execution Tree Viewer. In fact, the displayed execution-tree is the same as the one in Fig. 6.4, but with some differences concerning the point of view. This tree is intended to be seen as a three dimensional tree display, in slanted perspective. We are looking "from above" the general plane of the tree. This relief is used to compensate to some extent for the lack of information about the rule structure, as explained in Fig. 6.3.

Although we know that successful calls go down, we don't know *a priori* how to distinguish different call levels in the "flat" tree display (Fig. 6.4). It is impossible to guess that the `horsDoeuvre/1` and `dessert/1` calls belong to the same rule's body and that the `meat/1` and `dessert/1` calls do not. This is the purpose of the "non flat" tree view to give some clues about the level of every call, through the graphical depth of its related box (Fig. 6.5.)

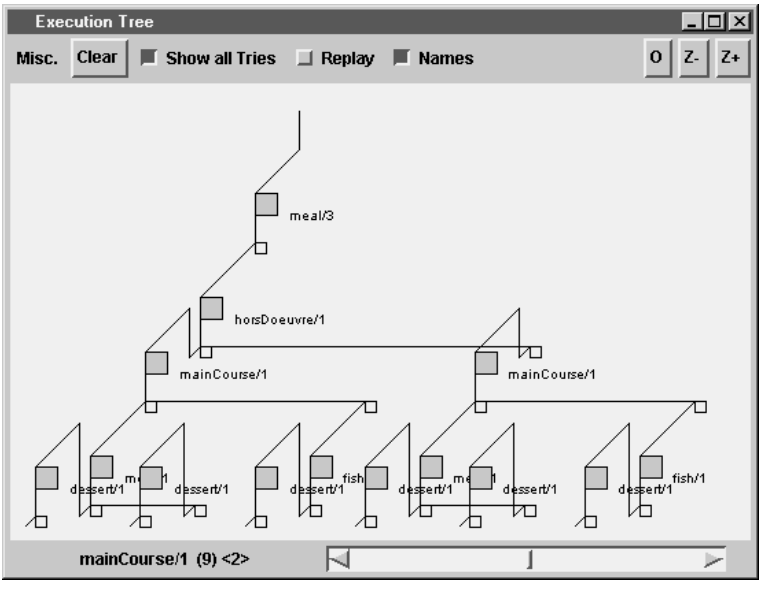


Fig. 6.5. An execution-tree in the Prolog IV Execution Tree Viewer (program `meal/3`)

For more examples, have a look at Fig. 6.6 to see the execution-tree of a deterministic program, and at Fig. 6.7 to see a more complex combinatorial program.

6.2.4 Textual Information in the Canvas

When the mouse is moved over a box, details about the box are displayed in the bottom left corner of the window. This information, derived from the box model, consists of

– the predicate indicator as *name*/*arity*,
– the rule number, if we are on a choice-node,
– the box number, for instance (11),
– the level of the call, for instance <3>.

Since this information corresponds to the Box Model Viewer print-out when it displays ports, the user is able to spot a given box on both viewers.

6.3 Viewer Functions Description

This section presents an overview of the most interesting Execution Tree Viewer functions, available from user interface buttons or menus.

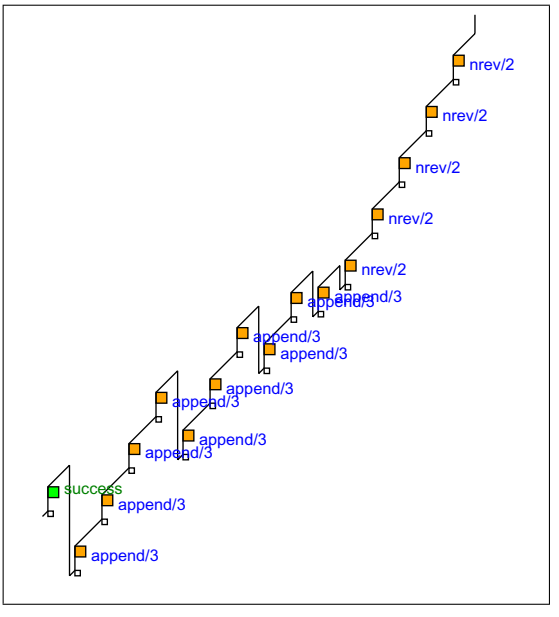


Fig. 6.6. Deterministic execution of a program ; in this example, the program is the well known `naive reverse` and the query is `nrev([1,2,3,4],L)`

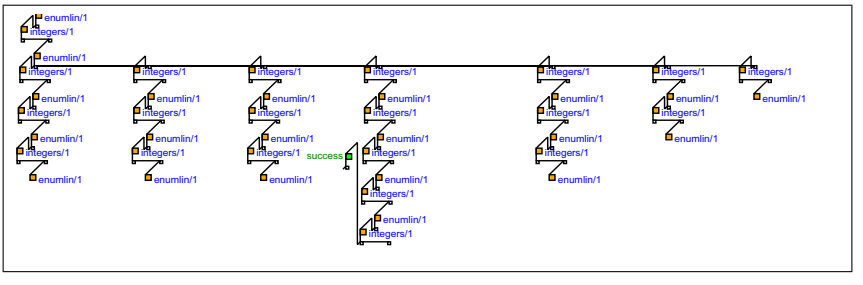


Fig. 6.7. Extract of non deterministic part of the execution of a program ; this example is the well known `send+more=money` combinatorial puzzle

6.3.1 Colouring Boxes

A cascade menu consisting of predicates names is built-up dynamically during execution. If the user associates a colour with one of these predicates, a chip of the chosen colour is displayed around all boxes with this predicate name, in the execution-tree. This enables the user to locate a given predicate in a big execution-tree or get an idea of the distribution of the whole set of calls of this predicate for profiling purposes (Fig 6.8 shows an example.) This is extremely useful in the case of big execution-trees, since names cannot be printed, and because, obviously, there are a lot of boxes. The user can select a few predicates to appear in some distinctive manner in the drawing. This is discussed specifically in [6.5], concerning the Prolog IV part.

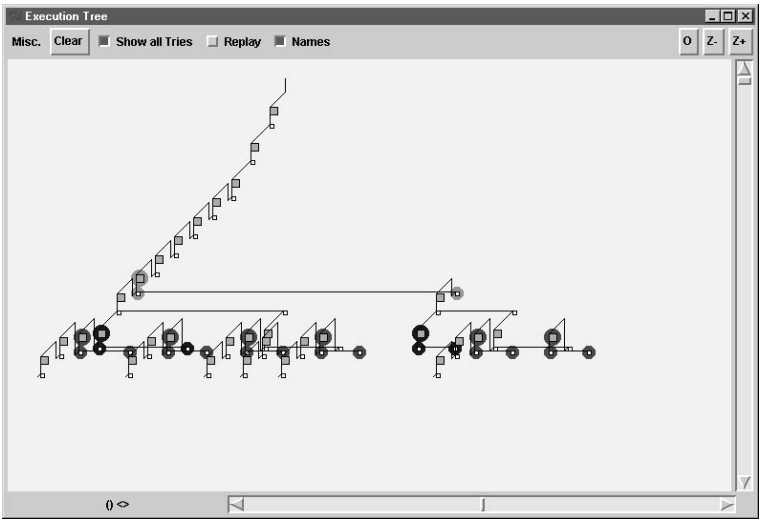


Fig. 6.8. An execution-tree with some coloured chips (pictured with grey levels.) Three predicates have been selected and given their own colour from a menu

6.3.2 Replay Mode

This is a running mode of the prolog debugger and the Execution Tree Viewer where

- the contents of the canvas are not destroyed when the next query is launched,
- the tree can be used to choose both a temporary break-point and its proof branch,
- the user is allowed to execute a goal which will run on the previously made tracks until the previously chosen break-point is reached.

Note: No more boxes are drawn during the execution, since it is assumed they are already displayed. See section 6.4.2 for more information on this topic.

6.3.3 "Replay-Left Mode" Option

If this option is checked, (and if the user is in replay mode,) execution of the next query will traverse the part of the tree to the left of the chosen branch. If it is not checked, only calls of the selected branch will be re-executed. This could be done safely if the program is pure-Prolog *i.e.* there is no call to predicates performing side effects (like I/O, asserts or global assignments) in the "left part" of the execution-tree with respect to the selected branch. Otherwise, this option should be left unchecked in order to ensure all calls (including calls to side effect predicates) are executed.

It may be important to know (about) the of the program that are "safe" since running a single branch is normally much faster than performing the full execution until the branch is reached, which consumes time trying known and expected dead ends or giving unwanted solutions —since we are debugging at an other place.

6.3.4 "Show all Tries" Option

This Boolean flag configures the viewer to enable or disable display of every attempt to match a call with a rule's head. If the Boolean flag is not set, only successful matching heads are displayed. If the Boolean flag is set, all heads are displayed, including non-matching heads.

Unsetting this flag will result in a significantly smaller execution-tree for the next user query to be launched. The execution-tree is narrowed in the left-right direction, (the OR axis). In example shown in Fig. 6.6, this flag has been unset, and so a narrow execution-tree has been drawn.

6.3.5 Miscellaneous

Several other controls affect the display in the canvas. Here are some of their functions:

- Making the origin of the tree visible in the upper left corner of the canvas.
- Scrollbars can be used to move around in the canvas to see hidden parts of the execution-tree.
- Zooming up and down. It is possible to scale up or down all objects and distances in the canvas. Scaling up enlarges items, and tree details are easier to understand, but less of the execution-tree is visible. Scaling down reduces the size of graphical objects, but more of the execution-tree can be seen.
- Displaying predicate names for all call-nodes. This is useful only if the current zoom factor is at a reasonable value. Otherwise, names are not displayed because the whole canvas would be unreadable.
- The Zoom Window is a feature providing a general view of the execution-tree and a detailed view of specific parts of this tree. In fact, rather than a zoom factor the Zoom Window uses the default size for boxes, which makes the local structure of the execution-tree readable and understandable. The Zoom Window can be activated by clicking a mouse button anywhere in the displayed execution-tree. This makes the zoom window appear, or changes its contents.

6.4 Working with the Debugger

First of all, remember that the debugger can only fully handle programs compiled with the "debug" option set. A program or predicate that is not

compiled with this option is seen as a single call: it is considered as a black-box. This feature is a great help when instead of a full detailed execution-tree the user wants to display some interesting predicates (from the user point of view at a given time.) Just (re)compile these interesting predicates with debug mode "on" to see them "expanded" on the Execution Tree Viewer at the next execution of the query. All other predicate (compiled with debug mode "off") calls then appear as a single box, with eventual choices going out from them. These choices can be seen as pseudo-rules, since it is not known at all where these choices come from precisely (lower levels are not visible). For instance, several enumeration predicates belong to the Prolog IV system, and as such are not compiled with debug mode "on". Each successful call to these predicates appears as if there was an indeterminate set of facts matching that call, hence the expression "pseudo-rule". Predicate `enumlin/1` gives an example of this in Fig. 6.7.

As a general remark, note that it is not mandatory to run all viewers to make them work. The ones not needed at a given time can be deactivated to avoid too much information being presented to the user and a loss of CPU time in displaying them (since a prolog-like engine, even in debug mode, produces information much faster than the operating system can display it.)

6.4.1 Setting a Temporary Break-Point

Clicking a mouse button on a box sets a temporary break-point on it. It also highlights the branch coming out of the top (the query goals) to this node (an example of this is shown in Fig. 6.9.) Either a call-node or a choice-node can be chosen as a break-point. Setting a break-point greatly enhances the execution control of the program being debugged, since it enables the user to specify a particular call-node (and thus a `CALL` port) to stop at[3]. This also enables "smart" replaying of program execution —where smart means "if possible, re-execute just what is needed to arrive at the selected box". Of course, in the case of deterministic programs, nothing smart could be done since the execution-tree is reduced to a single branch, but in most cases, useful prolog programs are not deterministic and some parts can sometimes be skipped in the context of debugging.

6.4.2 Replaying the Execution

After a temporary break-point has been set in the previous step, checking "Replay" mode in the Execution Tree Viewer changes the behaviour of the viewers: provided the query is the same as the one which has drawn the execution-tree, launching it will replay the execution "using" the previously displayed execution-tree. If "replay-left" mode is off, only the calls of the

[3] The other Prolog IV viewers only allow spy-points *i.e.* setting break-points on *every* `CALL` port with a given predicate name.

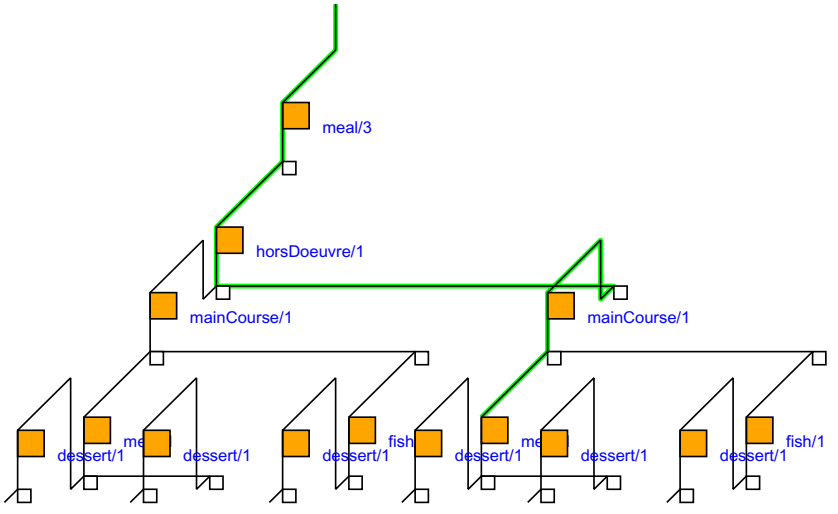


Fig. 6.9. A branch has been chosen by clicking a node

selected branch are executed; or else the whole of the execution-tree located to the left of the selected branch (included) is executed. In "Replay" mode no other boxes are drawn in the Execution Tree Viewer. If the user chooses to execute the program step by step, the current box is highlighted.

The user can choose a file containing the description of a previously executed and saved tree and load it in the canvas. Provided the same program is loaded and the same query is fired, one can use this execution-tree to perform a re-execution.

6.5 About Implementation

Essentially two kinds of work have been carried out to implement the Execution Tree Viewer. They can be briefly described as follows:

- the Prolog IV core debugger has been extended and now implements a new port and new debugger registers. Also, this core now generates special orders to be sent to the graphical front-end. These orders contain a description related to a subset of ports of the Prolog IV Box Model Debugger (not all ports are needed to build and draw an execution-tree).
- the user front-end has been built, (mainly the Execution Tree window) and the order-interpreter implemented (to decode core debugger orders, build boxes and connections between them, manage replay modes that monitor replayed execution, etc.)

The Execution Viewer is implemented with language TCL and its graphical library TK [6.7], as are the other components of the Prolog IV graphical environment (and thus the other debugger viewers.)

6.6 Conclusion

We have presented an extension to the Prolog IV debugger. This new tool complements the other viewers of the debugging environment and makes the whole debugging process easier by simplifying tasks as complex as re-executing part of an execution-tree or setting a particular break-point. Display of the execution-tree gives precious information about the nature of the problem being solved, by showing how this tree grows and evolves over a period of time, particularly when the program to debug needs implementation of good enumeration strategies. In this context, the execution-tree viewer can be a tool that helps define and refine the enumeration strategies required to develop the best possible domain reduction based solving systems.

This viewer has been introduced in the Prolog IV environment and has been used on fairly big applications solving complex problems. [6.5] presents users' comments on this tool.

References

6.1 F. Benhamou. Heterogeneous Constraint Solving. In: Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139 . Aachen, Germany, pp. 62–76, 1996.

6.2 F. Benhamou and Touraïvane. Prolog IV : langage et algorithmes. In: JFPL'95: IV^e Journées Francophones de Programmation en Logique. Dijon, France, pp. 51–65, 1995.

6.3 P. Bouvier. DiSCiPl deliverable: Visual Tools for Debugging Prolog IV Programs (a chapter of the Prolog IV manual). Deliverable D.WP3.5.M2.2. DiSCiPl - Debugging Systems for Constraint Programming (Esprit 22532), 1998. `http://discipl.inria.fr/`

6.4 A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix-Marseille, 1973.

6.5 T. Cornelissens et al. General report on assessment of the tools. Deliverable D.WP1.3.M1.3. DiSCiPl - Debugging Systems for Constraint Programming (Esprit 22532), April 1999. `http://discipl.inria.fr/`

6.6 W. J. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In: F. Benhamou, and A. Colmerauer (eds.): Constraint Logic Programming: Selected Papers. The MIT Press, 1993.

6.7 J. K. Ousterhout. Tcl and the Tk Toolkit, Professional Computing Series. Addison-Wesley, 1994.

6.8 Prolog IV team. The Prolog IV Manual (reference and user manual). PrologIA, 1996.

6.9 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In: L. Naish (ed.): Proceedings of the 14th Int. Conf. on Logic Programming. Cambridge, pp. 286–300, 1997.

7. Search-Tree Visualisation

Helmut Simonis and Abder Aggoun

COSYTEC SA,
4, rue Jean Rostand,
F-91893 Orsay Cedex, France
email: {Helmut.Simonis, Abder.Aggoun}@cosytec.com

This chapter describes a visual tool for debugging and analysis of the search-trees generated by finite domain constraint programs. The tool allows to navigate in the search-tree in a flexible way and gives, for any node of the search-tree, a clear view of the current state of the program execution. The tool provides graphical representations of the form of the search-tree, of constraints and variables of the program and of the propagation steps performed after each decision in the tree. The debugger is used via a set of meta-predicates which annotate the search routine given by the user, which allows great flexibility in adapting the program to the needs of different users. The tool is now part of the CHIP constraint programming environment and covers important aspects both of correctness and performance debugging.

7.1 Introduction

In recent years, a significant number of applications have been developed using constraint programming (CP) technology [7.13] [7.15] [7.16]. The complexity of problems handled is increasing and improvement of the debugging facilities becomes an urgent task. Currently, the CP technology is largely lacking debugging tools and a debugging methodology to support users. This methodology is a key point because CP programs are, different from conventional programs, data-driven computation rather than program-driven. Typical finite domain programs are structured into three parts: variable definition, constraint statement and finally the search procedure. Debugging concerns all three parts, but special emphasis lies on the search procedure, as most real-life optimisation problems encountered in industry have a very large search space. To understand the effect of the search procedure on the search space there is a requirement for a novel visual tool which allows to perform an abstraction of the constraints, and which shows different views of the variables, constraints and the search space. At the same time, its use should be simple and intuitive and should not require major changes in the program under analysis. According to requirements collected from different users of CHIP and a study inside the DiSCiPl project [7.8], debugging tools should be usable at different levels of expertise, from a novice constraint programmer trying to understand how constraints work, to the expert programmer developing and debugging large applications, but also by the tool developer

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 191–208, 2000.

to understand existing and to help find new or improved propagation mechanisms. It is important not only to cover the aspect of correctness debugging, finding errors in the logical meaning of the program, but also to help performance debugging, improving the speed of a correct, but slow application. The concept of global constraints introduced in CHIP [7.2] [7.4] has drastically reduced the number of constraints needed to express a problem, and allows the programmer to focus more on heuristics for the search procedure. At the same time, new debugging requirements for these powerful abstractions have arisen. This chapter discusses the search-tree visualisation tool for CHIP [7.14] developed at COSYTEC in the DiSCiPl project. The chapter is structured as follows: We first review existing work on debugging tools for finite domain programs in section 7.1. We then give in section 7.2 a motivation why search-tree visualisation is an important aspect in the development of large scale constraint applications and present the overall working principle of the search-tree tool. This is followed in section 7.3 by a description of the programmer's interface used. In section 7.4 we present the different views the tool offers. In the last section 7.5, we describe the current state of development and further features, which are currently under development. Visualisation tools for the global constraints in CHIP are described in chapter 12, while chapter 13 gives examples and an analysis of the use of the visualisation tools presented here.

7.2 Related Work

Given the recognised difficulty of developing correct and efficient constraint programs, there is a surprising lack of work on debugging aspects in constraint programming. In most systems, debugging tools are based on a trace of the execution. This makes it very hard to extract general information about the search procedure, and also leads to much time consuming navigation through the trace in order to find the current point of interest. The paper of Meier [7.11] describes GRACE, a tool to visualise domains in the context of a normal box-model trace. It can also be used to follow (in a textual form) individual propagation steps in the trace. The tool can be extensively reconfigured by the user to execute user-written code at each step of the trace process. It takes advantage of the fact that the propagation engine is written in Prolog, so that modifications can be performed in the kernel. The paper of Schulte [7.12], describing the Oz Explorer, is the main influence on our work. The Oz Explorer provides a graphical interface to display and control the search. It is possible to collapse or expand parts of the search-tree in order to concentrate on interesting sub-parts. A major advantage over our system is the possibility to program new search methods with a small set of primitives of the concurrent language. On the other hand, it does not contain at the moment views on constraints or on propagation steps so that its capability to

follow the reasoning inside a search routine is somewhat limited. Very interesting work on visualising search has also been done from an OR perspective [7.10]. An early paper of Held and Karp [7.9] already shows search-tree displays very similar to the format used here. In the context of the DiSCiPl project, other debugging methods known in the logic programming field, like static analysis and declarative debugging are also considered for constraint programs [7.5].

7.3 Principles of Operation

In this section we will describe the assumptions underlying our tool, the basic principle of operation and some implementation details. First we describe the different debugging problems encountered and why we have chosen the search-tree as a good candidate for debugging.

7.3.1 Program Structure

Finite domain constraint programs typically consist of three parts, variable definition, constraint set-up and search. As the constraint solvers for finite domains are incomplete, we must choose among undecided alternatives (for example alternative values for variables) until a ground solution is found. The search procedure typically is based on depth first, chronological backtracking, but partial search methods [7.3] are becoming more and more popular. In any case, the search will consist of a number of nodes corresponding to states of the constraint store. Children of a node will be created by selecting an open decision and branching on possible alternatives. For the developer of an application, the most interesting aspect is the debugging of the search process, on which we will concentrate in the current chapter. Other aspects of debugging are for example the constraint set-up. If it fails, it is often quite simple to find the cause. In more complex situations, we must detect a difference between the intended model of the problem and the model described in the program. The resulting specification debugging is quite difficult and mostly a manual process. In addition to these aspects of constraint programming, there are usually program parts concerned with data gathering and preparation. These pose "conventional" debugging problems.

7.3.2 Symptoms

Problems in the search part will normally create one of the following symptoms:

No solution: A solution exists (for example created manually) but the search procedure fails. A good test will be to feed a known solution into the constraint solver, creating a "missing answer" problem.

Wrong solution: A solution is found but it is judged infeasible by the user. This may be caused by a bug in the application program or the constraint solver, but most often is linked to an incomplete or incorrect specification. A good test is to run the constraint program as a test only. This will often detect if the problem lies in the constraint engine.

No answer in given time: The system is backtracking in a large search space, without finding a solution and without enumerating all choices in a given time limit. It is not clear if a solution exists at all. This is a good candidate for search-tree debugging.

Answer found, but performance not satisfactory: This is the field of performance debugging. A change in the search strategy may find a solution more rapidly, redundant constraints may improve propagation, a change in the specification may make the problem harder (more propagation) or simpler (more solutions). Again, this is an area where search-tree debugging is most useful.

7.3.3 Operating Mode

We rejected the use of an off-line tool for visualisation, although these have been quite successful in visualising logic programming languages [7.6]. With constraint programs, too much information, i.e. all constraint propagation steps and all modifications of domains, would have to be stored, creating a very large trace file. Our tool works in co-operation with the constraint solver, replacing the usual labelling routine. In a first phase, the user defined search routine is simulated, and the structure of the search-tree stored in search node objects. These objects record the parent of the current node, the decision which is selected, and the branch which is taken. In the most basic case, the decision concerns a variable to be assigned and the choice is one particular value which is chosen. The simulation of the search procedure stops when one of the termination criteria is met:

- Number of solutions found.
- Number of failures in the search.
- Number of nodes explored in the search-tree.
- Execution time limit exceeded.

The search tee tool will stop when any of these criteria are satisfied and will display the nodes that have been explored so far, even if the user-defined search procedure did not find a solution.

Once the information about the tree is complete, the system backtracks to the state at the root of the search-tree, the tree is displayed graphically and the user can interact with the system. When a node of the tree is selected, the system will re-create the state of the computation at this node, following the path from the tree root to the node and re-enacting the decisions taken on this path. This will lead to a state of the constraint store which corresponds to the original state at this node. Navigating between nodes will cause backtracking

to the root state and repeated re-enacting of the solution on the path to the selected nodes. The system must work on the full functionality provided in the CHIP environment:

– large sets of domain variables with possibly large domains.
– all basic and global constraints.
– user-defined constraints (co-routines, demons).
– predefined and user-defined heuristics.
– meta-heuristics (partial search) .
– optimisation meta-predicates (`min_max`, `minimize`).

In the current implementation there are restrictions on which type of constraints can be used to branch on in the search. In the following, we assume a value choice branching, i.e. at each node we choose some value for a variable.

7.3.4 System Requirements

The search-tree tool is conceptually quite simple, but a number of primitives must be available for an efficient implementation.

– In order to record the changing current parent in the tree, a form of trailed assignment [7.1] is useful. This assignment restores the previous value of the parent variable whenever some backtracking occurs.
– Both constraints and variables must have unique identifiers which can link source-level text to implementation objects.
– To follow propagation, it is required to have access to all constraints linked to a variable at all times.
– Meta-programming is very useful to simulate the user-written search procedure inside the search tool.
– In order to understand which actions are performed by propagation, we have chosen to implement *propagation events*, a log of all propagation actions (wake, update, state-change, fail) that can be accessed in an asynchronous way. We can thus access the sequence of propagation steps as data without deep changes to the propagation engine. These propagation events can also be very useful for statistics and meta-programming.

7.4 Interface

The interface to the visualisation tool should be as simple as possible, while allowing the user to control all aspects of the display. For a novice user with small example programs it is possible to extract automatically all variables and all constraints and to display the complete search-tree generated by a standard search routine. For more complex programs, the user must annotate the source code to indicate which variables and which constraints should be displayed. The user must also annotate/rewrite custom search procedures to

indicate which choices should be displayed in the search-tree. Two methods for marking variables are possible. One approach consists in marking in the source code each variable which should be handled in the visualisation tool. The other approach consists in passing all variables that should be handled as arguments to a search-tree procedure. In the CHIP visualisation tool we use the second alternative. The user should be able to indicate that all constraints should be included, or should be able to individually mark/unmark constraints. This can be done in the form of annotation around the constraints in the source code. By default, all constraints which use the selected variables are displayed.

For the search part, the easiest case is the use of the built-in search procedure `labeling(Terms, SelectedArgument, VariableChoice, Value_choice)`. The labelling routine performs variable selection and value assignment in a value choice scheme, where heuristics and choice functions are given by the user. In order to visualise this search scheme, the call to `labeling/4` must be replaced by a `search_labeling/4` predicate which takes the same arguments and which automates the generation of the search-tree. In addition, the `search.pl` library must be loaded.

For user defined procedures two annotations are required. One is a wrapper around the search part of the program, which indicates when the search starts and when it ends. This predicate, `search_start/2`, takes a list of domain variables as its first argument and the call to the search routine as second argument. The other predicate is a wrapper around each choice inside the search routine. The user can decide to see each choice individually or combine several choices into one. This gives additional control over the presentation of the search-tree. If several choice points are combined, the displayed depth of the tree decreases, while the width increases. The predicate `search_node/3` takes three arguments, the variable which is affected, the index of that variable in the list of variables passed as first argument to `search_start` and the call to the choice predicate. In figure 7.1, we show the modifications required to an application in order to use the search-tree library. The program is the well-known ship-loading problem described in [7.2].

The key to success for debugging large-scale applications is the restriction of the displayed information to important parts. A large scale, industrial constraint application will often require several thousand lines of CHIP code, using a few thousand variables and several dozen global constraints, together with hundreds of simple constraints [7.13]. The user must be able to control the volume of information both when generating the search-tree and inside the visualisation tool. At each point, it is possible to reduce or to increase the amount of displayed information under user control.

```
?-lib search.

run(Start, Upper, Last):-
    data(Nr,Dur,Use),
    length(Start,Nr),
    Start :: 0..Last,
    Limit :: 0..Upper,
    End :: 0..Last,
    precedences(L),
    set_precedences(L,Start,Dur),
    cumulative(Start,Dur,Use,unused,unused,Limit,End),
    search_start(Start,min_max(labeling(Start),End)).

labeling(L):-
    search_number(L,Merge),
    labeling(Merge, 1, most_constrained, assign).

assign(t(X,N)):-
    search_node(X, N, indomain(X)).
```

Fig. 7.1. Programmer's Interface for Search tool

7.5 Views

In this section we describe the different views of the search-tree visualisation tool, i.e. different graphical representations of the search, the variables and the constraints of the problem.

7.5.1 Types of Views

The visualisation tool provides a number of views into the search procedure. These views are based on four concepts:

State information is displayed for one particular state of the search-tree, e.g. the domains of all variables at a particular node of the search. Moving from one state to another updates all information.

Path information is displayed for a path in the search-tree from the root node to another node. This shows the change of constraints and variables over all choices leading to the selected node.

Tree information is displayed for all nodes below the current node. It can be shown either as the intersection or as the union of the information provided by all nodes in the sub tree. This concept is used for advanced features like propagation lifting, described at the end of the chapter.

Evolution information is displayed for all nodes in the search-tree explored before the current node. This concept is useful for statistics on failures, calls of predicates, number of propagation steps. Some of the information may be easily available for the complete search-tree, but may be quite

difficult to obtain for each node in the tree without re-running the search procedure.

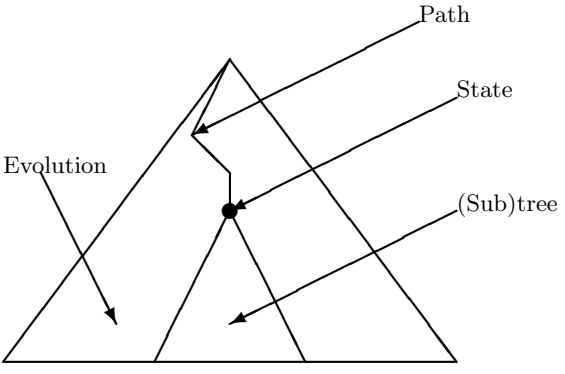


Fig. 7.2. View Concepts

The diagram in figure 7.2 illustrates the different view concepts. We will now describe the different views and show how they relate to this general classification.

7.5.2 Tree View

The search process is represented in tree form with each connection from parent to child indicating a separate choice. The tree view is used to analyse and to navigate through the search space. Figure 7.3 shows a small part of such a search tree. It is taken from the search-tree generated by the ship loading program shown above, as are the other screen pictures used in this section.

Nodes. The tree consists of different types of nodes, which are detailed below. Nodes are colour coded with a user modifiable colour scheme. A typical scheme would use colours to show the number of alternatives in each interior node or would show the different types of nodes in different colour, as actually shown in figure 7.3. If a node is displayed crossed-out, then all available choices for the selected variable have been explored, i.e. there are no more alternatives to be tested. The text written inside a node can display a number of different values depending on a user selection:

– the name of the variable,
– the level in the tree,
– the index in the variable list,
– the value selected.

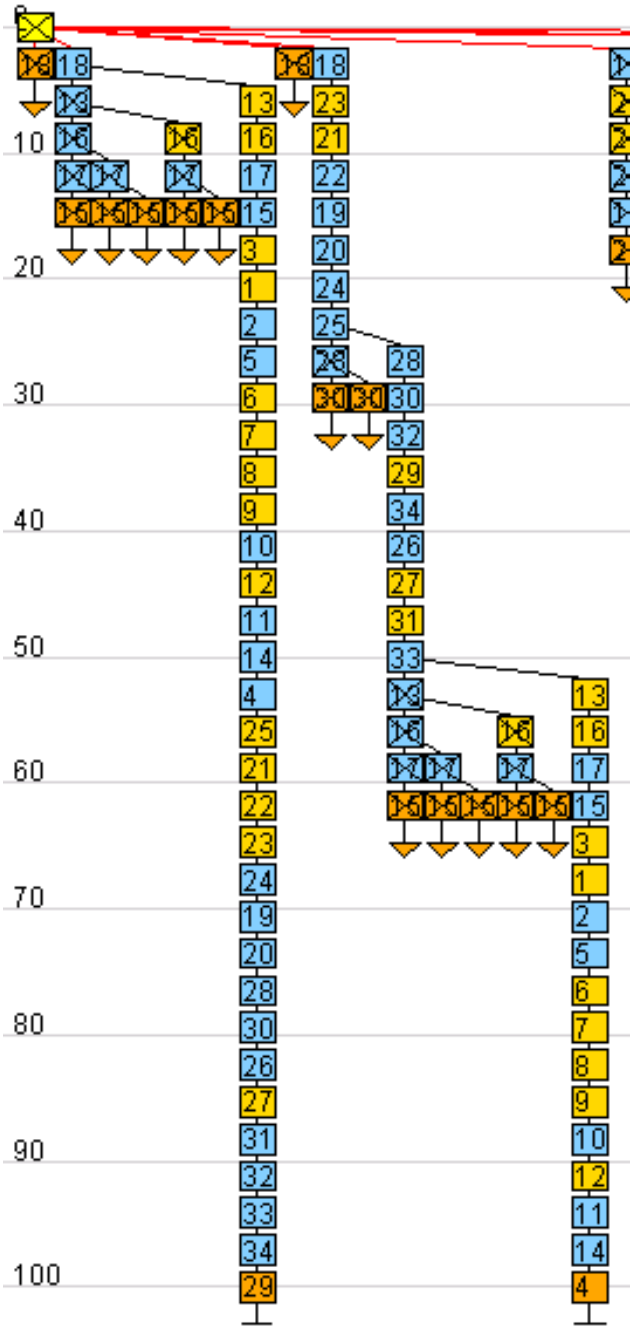


Fig. 7.3. Tree View

The root node corresponds to the state of the constraint system after the constraint set-up, before the enumeration process. At this point it is possible to see the domains of all variables after the initial constraint propagation. The interior nodes correspond to states in the search-tree where not all choice points have been explored and the constraint system has not failed.
Failure leaf nodes are displayed when the constraint system fails. Failure nodes can be shown in two ways: the more explicit representation shows each failed choice in the search tree. If for example, a indomain call tested 10 different values and each of them failed, then this representation would display 10 failure nodes. This has the advantage that there is a well-defined unique failure cause, i.e. one constraint that failed during propagation. The disadvantage is the possibly large number of failure nodes. A more compact representation shows one node for every decision predicate (like indomain) which failed. One failure node here abstracts possibly many failures, where individual values have been tested and the constraint system failed. The advantage is the more compact form of the search, the disadvantage is the difficulty to explain the propagation which led to the failure. In the current implementation only the second representation is implemented.
Success leaves are reached when all variables in the problem have been assigned and a solution was found. Note that quite often the last steps of the search procedure are induced by constraint propagation, i.e. the last real choice in the search may be several levels above the success leaf and all nodes below this choice are deterministic, i.e. the variables selected already have been assigned by constraint propagation.
In order to present the search-tree in a more compact form, it is useful to compact failed sub-trees into a single node, a failure tree. This tree is marked with the number of failure leaves it contains. The user can interact with the system to collapse or expand parts of the tree by hand. The system also has options to automatically collapse all failure trees or to expand all nodes. Similar to failure trees, the system can display several solutions as one success tree node. Again, the system indicates how many failure nodes and how many success nodes it contains. The user can collapse/expand any node to the corresponding failure or success tree.

User Interaction. The user can zoom to any part of the search-tree by either using scrollbars and/or selecting an area in the current display area with the mouse. The user can navigate through the search-tree from node to node by selecting nodes with the mouse and can collapse the sub-tree originating at any node or expand a currently collapsed sub-tree. Selecting a node sets the current state of the search process to the state represented by the node. This requires to re-instantiate all variables on the path from root to the selected node to obtain the path information, which will re-run all constraint propagation on this path. The system can show information about the selected node, like the level of the node in the search-tree and the index of the selected variable.

Phase-line Display. There is another way to display the search tree. Instead of displaying links for the parent-child relation, we can link all nodes which assign the same variable, without changing their placement in the display. With a static variable selection order, all those nodes are at the same depth of the search-tree, so that the phase-lines are straight lines. If the variable selection is very dynamic, then the phase-lines will indicate at which level some variable is assigned. Figure 7.4 gives an example of the phase-line display. As we will see in chapter 13, the phase-line display is also very useful to compare the variable selection order in two different heuristics.

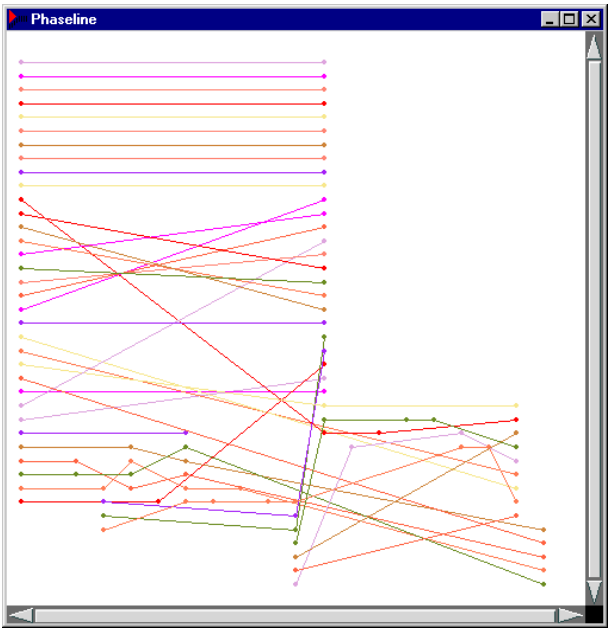


Fig. 7.4. Phase-line display

7.5.3 Variable Views

The different variable views are intended to help understand the impact of the search procedure on the different domain variables.

Update View. This view shows the change of the variables on the different steps on the current path from root to selected node. In x direction, all variables are shown, in y directions from top to bottom the levels of the search-tree. Each entry in this table marks the change of the variable with different colours. A typical example is shown in figure 7.5, showing the update view for the first success node of the search-tree.

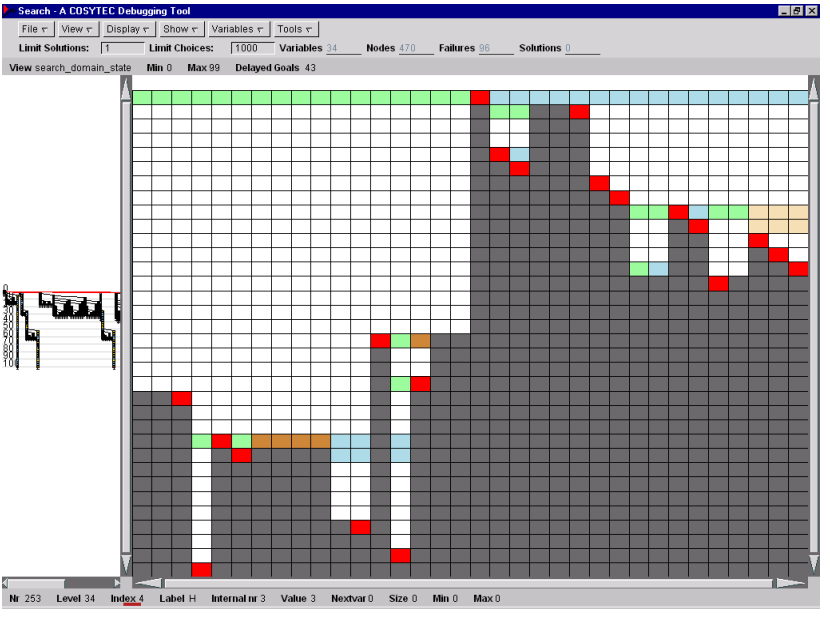


Fig. 7.5. Variable Update View

The following types of updates are recognised and displayed in different colours:

- variable is assigned to a fixed value in this step by search procedure (black).
- variable is ground (dark gray).
- min and max are updated (light gray).
- min is updated (light gray).
- max is updated (light gray).
- size is changed, i.e. interior value was removed (light gray).
- variable is unchanged from previous step (white).

Selecting a cell in the view will show the level and some information about the selected variable in the information display line. This view gives a good indication on how much propagation occurs in the program and which variables are influencing other variables. In the display above, we can see that only a limited amount of propagation is happening after the first variable has been assigned.

State View. This view shows the domains of all variables at a given state. In x direction, values in the domain are shown. Each line corresponds to a variable. The entries in this view show which values are currently in the domain of the variables. They can also show which values were removed in the last assignment step. Figure 7.6 shows the variable state view after the first variable has been assigned.

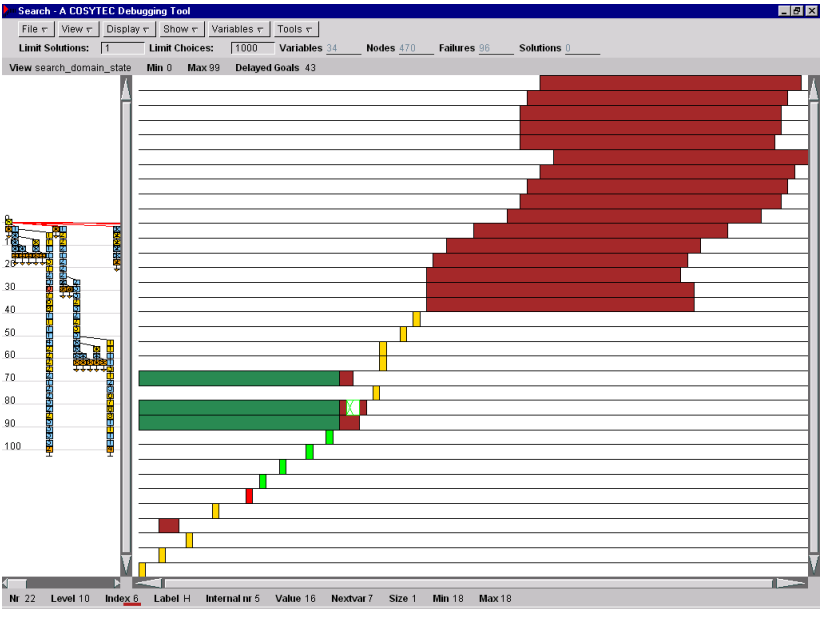


Fig. 7.6. Variable State View

For large domains, a textual representation of the domain may be more compact than a graphical one. It is also possible to restrict the display to a user-defined interval of values. Selecting a cell in the display will indicate which variable and which value were selected. The state view can be used to understand what information is available at a given point of the computation. We can see for example which values have been used more often than others in the current (partial) assignment.

7.5.4 Constraint Views

The constraint views show either the complete constraint network of the program, or show the evolution of particular constraints within the search process. The views are useful to understand the problem structure and the overall constraint reasoning. The general constraint views can be used for all types of constraints, others are specialised for the global constraints in CHIP. We only discuss the general views in this chapter, special visualisers for global constraints are discussed in chapter 12.

Incidence Matrix. All constraints can be shown as a constraint variable incidence matrix (see figure 7.7). In x direction, all variables are displayed, in y direction, all constraints are shown. If required, this matrix can be compressed by grouping constraints into lines which count occurrences of variables. This display gives an indication of the impact of constraints on variables,

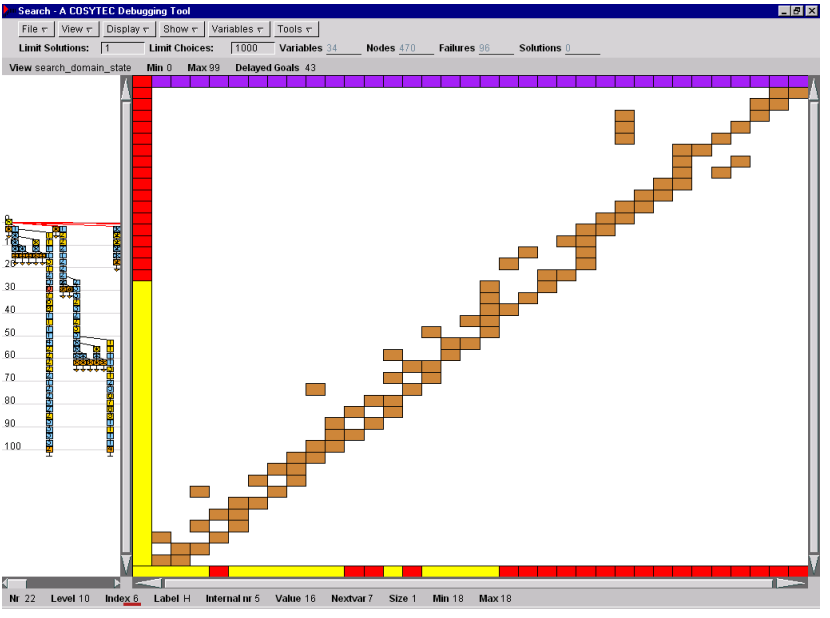


Fig. 7.7. Constraint Incidence Matrix

which is also used for the variable selection in certain strategies. Figure 7.7 shows the incidence matrix for the ship loading problem. The different types of constraints are colour-coded; in this example we find a number of binary inequality constraints and at the top a single cumulative constraint which expresses a resource limit. The use of global constraints makes the incidence matrix feasible, as there are not too many constraints to be displayed. For a selected state of the search-tree, we indicate in red lines on the left and at the bottom which variables and constraints are still alive, i.e. have not been assigned (resp. solved) yet. Selecting a line in the display prints the textual representation of the constraint in the text line. The incidence matrix gives a good, compact view of the inter-relation of variables and constraints. Hidden structures and symmetries can often be recognised in this view.

Update View. This view shows the activity of the constraints on the path from root to the current node of the search-tree. In x-axis, all constraints are shown, in y-axis, the different levels of the search-tree. An entry shows what happens to a constraint at this level in the search-tree. Different colours are used to encode whether

– the constraint is woken.
– the constraint causes some domain update (min, max, remove).
– the constraint binds some variable.
– the constraint is solved.

If more than one condition applies, the stronger (lower) one is displayed. This view is useful to see which constraints contribute to the propagation, e.g. to see if some redundant constraints actually contribute to domain reductions.

Constraint Count. This simple view shows a diagram with the level of the search-tree on the x-axis and the number of active constraints on the y-axis. This display shows the progress of constraint propagation as a graph.

In addition to these general constraint views, there are specific visualisation tools provided for the global constraints [7.2] [7.4]. Global constraints work on sets of variables using multiple propagation mechanisms for deduction. For each of these constraints, one or multiple graphical representations can be used to display the information available to the constraint. These visualisation tools are described in chapter 12 of this book.

7.5.5 Propagation View

The propagation views are used to understand the propagation for one assignment step. One assignment may lead (in a large problem) to several hundred constraints being woken and re-woken several times. This view is for the advanced user who tries to understand the interaction of the different constraints with each other. This display shows all propagation steps resulting from the current assignment. The view displays variables in x direction, and propagation steps in y direction from the top to the bottom. Each line of the display corresponds to one constraint. In each line, all variables which are used in the constraint are marked. Any variables which are updated are shown in a coding according to their update type. It is possible to restrict the display to show each woken constraint only once in the propagation view and to ignore all constraints which are woken, but do not further restrict variables. Selecting a line in this view will indicate the constraint and variable chosen. Figure 7.8 shows the propagation view of one step in the ship loading example. Setting one variable to a particular value leads to a series of updates via inequalities (each affecting two variables) and cumulative (affecting all constraints). The propagation view shows all steps of the propagation and their effect. For some choices, very little propagation will occur, some others will lead to hundreds of propagation steps. The user can zoom on the display to follow the propagation step by step.

For some purposes, the view shown is not detailed enough. For global constraints in particular, we can use *propagation events* to capture the reason for individual updates of the variables. Propagation events record all waking, domain updates and failures of constraints together with information about the variable which caused the event, the variable that was affected, the details of the modification and the particular method used inside the constraint. The propagation events are written into a log, which can be retrieved later on. This information is available in the propagation-event view, which lists in a textual form all propagation events caused by some step in the search

procedure. The propagation step display compresses the same information, it does not show for example the details of a domain modification.

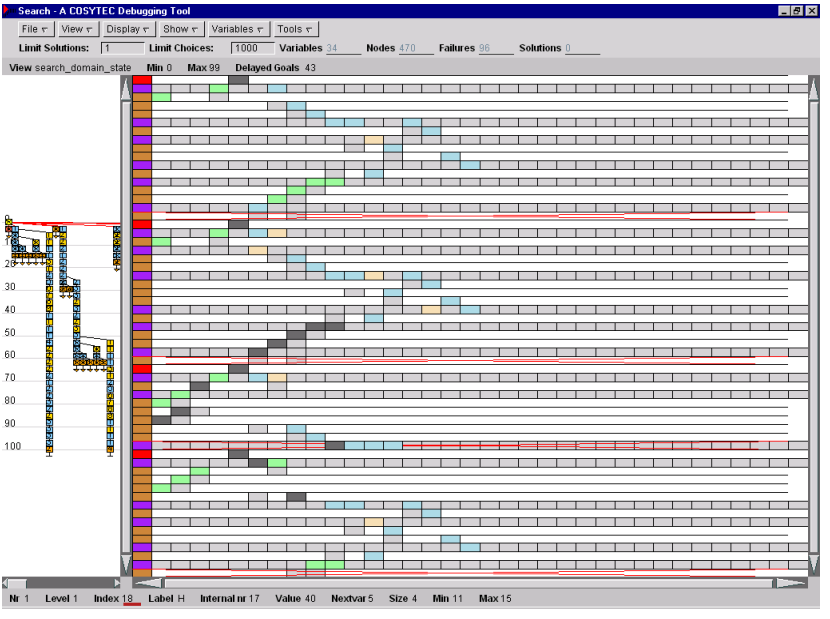


Fig. 7.8. Propagation View

7.6 Current State and Further Development

A first version of the visualisation tool has been implemented and is available as part of the current CHIP 5.2 release [7.7]. It allows the handling of small to medium sized problems (up to 500 variables). So far, neither the overhead when executing the search procedure nor the time to restate the variable bindings when selecting a node are causing performance problems. Besides extending the scale of problems that can be handled, we want to further improve the functionality. Some future extensions are listed in the following paragraphs.

– Improved display of isomorphic sub-trees: At the moment, a middle click on a node will find isomorphic copies of the tree under the selected node in the complete search-tree. We distinguish situations where variable and value are the same and situations where only the selected variable is the same. We want to improve this mechanism to automatically find all sub-trees for which isomorphic copies of a certain size exist. This is very helpful in

understanding thrashing behaviour, where the same constraint reasoning is applied multiple times.
- Why explanation for value removed from the domain of a variable: This module will find the level in the search, and the first constraint responsible for the removal of a value from a domain. While this not really gives an "explanation" for the removal, it can be used to better understand the constraint reasoning.
- Propagation lifting: If a value is removed in all leaf-nodes underneath a given node which has been completely explored, then the value can be removed at this node. Indicating such variables and values can be a powerful tool to discover missing constraint propagation.
- Failure analysis: The propagation events provide all primitives to explain failures from constraint propagation. Starting backwards from a failure, causal event links of relevant constraints can be built connecting the failure to some decision in the search-tree. Statistic analysis can also be used to understand which constraints detect the failure with a view to performance improvement.
- Tuning of Method Selection: The global constraints use a large variety of methods to deduce new information at each step. Not all methods will be useful for all problem types. We can use the constraint views together with propagation events to determine for a given problem which methods perform most of the propagation work. This would allow to (semi-) automatically tune the engine to handle particular classes of application problems with a restricted set of methods and a corresponding gain in efficiency.

7.7 Conclusion

In this chapter, we have presented a visual analyser for the search tree generated by finite domain programs in CHIP. The tool allows to navigate in the search-tree generated by the search procedure using different views for variables, constraints and propagation. The tools is aimed at all users of the CHIP system, providing different degrees of sophistication for the beginner and the experienced programmer alike. First experience with the tool has already shown its usefulness also for didactic purposes, helping to understand how constraint propagation and search work. A relatively small set of primitives is required, so that the analysis tool can be designed and modified without a deep integration with the problem solver.

Acknowledgement

We gratefully acknowledge the influence from many discussion with the CHIP team, in particular E. Bourreau and N. Beldiceanu, and with our partners in the DiSCiPl project, in particular the group of M. Hermenegildo at UPM.

References

7.1 A. Aggoun and N. Beldiceanu. Time Stamp Techniques for the Trailed Data in Constraint Logic Programming Systems. In Actes du Seminaire 1990 - Programmation en Logique, Tregastel, France, May 1990.

7.2 A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems. Journal of Mathematical and Computer Modelling, Vol. 17, No. 7, pages 57-73, Pergamon Press, 1993.

7.3 N. Beldiceanu, E. Bourreau, P. Chan, and D. Rivreau. Partial Search Strategy in CHIP. 2nd International Conference on Meta-heuristics, Sophia-Antipolis, France, July 1997.

7.4 N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.

7.5 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyǹski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In Proceedings of the Third International Workshop on Automated Debugging-AADEBUG'97, Pages 155-170, Linköping, Sweden, May 1997.

7.6 M. Carro, L. Gomez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. Proc. ICLP93, Budapest, Hungary. The MIT Press, Cambridge, MA, 1993.

7.7 COSYTEC. CHIP++ Version 5.2. Documentation Volume 6. Orsay, 1998.

7.8 M. Fabris et al. CP Debugging Needs and Tools. In Proceedings of the Third International Workshop on Automated Debugging-AADEBUG'97, Pages 103-122, Linköping, Sweden, May 1997.

7.9 M. Held and R. Karp. The Travelling Salesman Problem and Minimum Spanning Trees: Part II. Mathematical Programming 1, pp 6-25, 1971.

7.10 C. V. Jones. Visualization and Optimization. Kluwer Academic Publishers, Norwell, USA, 1996.

7.11 M. Meier. Debugging Constraint Programs. In Principles and Practice of Constraint Programming, page 204-221, Cassis, France, September 1995, Springer, Lecture Notes In Computer Science 976.

7.12 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. Proceedings of the Fourteenth International Conference On Logic Programming, Leuven, Belgium, pages 286-300. The MIT Press, July 1997.

7.13 H. Simonis. Application Development with the CHIP System. Proc Contessa Workshop, Friedrichshafen, Germany, Springer-Verlag, LNCS, September 1995.

7.14 H. Simonis. The CHIP System and its Applications. Proc. Principles and Practice of Constraint Programming, Cassis, France, September 1995.

7.15 H. Simonis. A Problem Classification Scheme for Finite Domain Constraint Solving. Proc workshop on constraint applications, CP96, Boston, August 1996.

7.16 M. Wallace. Survey: Practical Applications of Constraint Programming. Constraints, Vol. 1, Nr1-2, pp 139-168, September 1996.

8. Towards a Language for CLP Choice-Tree Visualisation

Christophe Aillaud[1]* and Pierre Deransart[2]

[1] Gemplus
Avenue Pic de Bretagne, Parque d'activité de Gemenos,
BP100, F-13881 Gemenos cédex, France
email: Christophe.Aillaud@gemplus.com
[2] INRIA-Rocquencourt, Projet Contraintes
BP 105, F-78153 Le Chesnay cédex, France
email: Pierre.Deransart@inria.fr

This chapter describes an approach to performance debugging of constraint programs by means of the visualisation of selected parts of the computation space. The computation space is represented by choice-trees, a notion which is commonly used to describe the search in constraint solvers. We propose a language for selecting the information to be visualised and present an algorithm to construct the corresponding view (it is a pruned choice-tree) in a form which is unique and which preserves the relative positions of the selected nodes and arcs. The originality of the approach lies in its genericity (different views may be displayed simultaneously to analyse the same computation space) and the way to specify the views, in particular, using properties of the program at hand. This allows to define new views of one execution without updating the program at hand nor re-executing it.

8.1 Introduction

This chapter describes an approach to performance debugging of Constraint Logic Programs (CLP) based on display of simplified views of the computation space. In constraint programming over finite domains, the search for a solution can be represented by a choice-tree which includes all the computation paths and all the choice points corresponding to domain splitting of the finite domain variables. Choice-trees may look very differently, depending in particular on the search strategy. In CLP the constraints are additionally structured by predicates and a computation is modelled by the notion of search-tree, developed in logic programming [8.6, 8.10, 8.13]. In CLP, search-trees correspond to and encompass choice-trees.

Choice-trees (and search-trees) are usually much too large to be fully displayed. Pruning them, in order to extract some relevant computation steps only, is a way to build simplified views of the search space, with the hope that the obtained views will help analysis of the computation complexity and of the used strategies, and therefore to improve program and performances.

* Christophe Aillaud contributed to this work during his stay at INRIA-Rocquencourt

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 209–236, 2000.

Different tools have been proposed for search space visualisation which focus on dynamic constraints display (like GANTT Charts) [8.9] or interactive restricted [8.1, 8.14] or unrestricted search space display (Chapters 6 and 9). All existing systems have predefined views of the search space with some collapsing functionalities which can be activated from the window in which the search-space is visualised. In [8.15] there is a possibility to specify new search space representations, but the user has to define and manipulate himself such objects.

Our approach is different. It assumes familiarity with the notions of search- or choice-trees. These objects have been studied for many years and even if their definition may vary, depending on the authors, the general idea is quite common. We follow here the seminal search-tree definition of Lloyd [8.13]. Search space visualisation must be based on a well understood paradigm and therefore search-trees are good candidates. However, even if the idea is clear, search-trees are usually much too large and too complex to be of any practical help. There is no hope (in practice) to recognise any significant pattern, just looking at the whole tree (consider for example the tree of calls of nondeterministic multi-recursive procedures). We therefore want to avoid to have to manipulate search-trees as objects, nor to have to look at any complete search-tree[1]. This is possible in CLP, because nodes and arcs in the search-tree correspond to elementary computation steps which can be easily identified in the text of the program.

In fact (this will be comprehensively explained in the chapter) every step of computation (hence every node of the search-tree) corresponds in CLP to the execution of a predication which can be identified by a property, i.e a predication and a relation between its arguments[2]. Some computation steps (arcs in the search-tree) can be identified by two predications in a program clause. It is thus possible to specify restricted parts of search-trees, just by looking at the program at hand and specifying computation steps by predicate properties.

Let us for example consider the very simple deterministic Prolog program

```
:- p(10).
p(X) :- X1 is X-1, p(X1).
p(0).
```

To each of its computation step corresponds a call to `p(N)` with $0 \leqslant \mathtt{N} \leqslant 10$. A way to select exclusively the computation steps number 4 to 8 is to use the property of the predicate `p/1` expressed by `p(X)`: $4 \leqslant \mathtt{X} \leqslant 8$.

This chapter presents ideas needed for constructing a support tool for definition and display of views of the search-tree in the form of pruned trees. It addresses two problems: how to reduce the original search-tree wrt a gi-

[1] This may however be useful with very small execution or for pedagogical purpose.
[2] In practice, this may oblige to modify some predicate in the program with additional arguments.

ven selection of arc and nodes and how to define such selections so that the pruned tree can be specified and displayed without modifying the program at hand. The proposed solution to the first problem is reflected by a natural concept of minimal pruned tree (Definition 8.3.3). For the second problem we sketch a specification language. We show by examples that it is sufficiently expressive for defining practically relevant selections. We also discuss its implementation principles and a prototype visualisation tool. We show that the implementation does not require changes in the analysed program.

After an introduction of the basic elements of the operational semantics of CLP and of state properties (Section 8.2) this chapter is organised in three main parts. Section 8.3 defines pruning of ordered tree in a manner which is independent from their meaning (it applies to any ordered labelled tree). It gives a declarative definition of the pruned tree corresponding to a subset of selected nodes and arcs, and describes how the pruned tree can be computed. This gives a declarative flavour to the notion of view. In Section 8.4 a language for node and arc selection, using state properties, is sketched. Finally Section 8.5 deals with implementation and examples.

8.2 CLP: Syntax, Semantics and State Properties

In this section we recall the syntax and the semantics of CLP programs, with focus on operational semantics. In this semantics computation states are characterised by "constrained predications" which will be observed by their properties.

8.2.1 Syntax

A CLP program is built over finite sets of function symbols, predicate symbols and denumerable set of variables. Among the predicates we distinguish a special class of constraint predicates[3], including the standard equality. The function symbols are divided into constructors and function names.

The atomic formulae constructed with the constraint predicates will be called basic constraints. The other atomic formulae will be called predications[4].

A program is a set of clauses of the form $h \ :- \ \{c|\ b\}^*.$,where c is a basic constraint and h and b are predications. Predicates include system predicates of the existing CLP systems.

A *unit clause* is a clause in which the body has constraints only. A clause with empty body can be just a head.

A goal is a clause with no head.

[3] Constraints are a subset of the built-in predicates in a CLP system.
[4] ISO Prolog notation for "atom".

8.2.2 Constraint Domains

The constraints of a CLP program are interpreted over some structure D. The language of constraints together with the interpretation form a constraint domain. In practice a single CLP language may include several domains and some interaction between them may be possible. Typically the CLP languages make it possible to combine the use of interpreted function symbols belonging to some specific constraint domain and term constructors from the Herbrand constraint domain. Domains of variables can be inferred from the constraints in which they occur.

Example 8.2.1. Here is a CLP(FD)[5] program in Calypso[6]

```
Pr1   sp1(N,[P|T],[Q|R]) :- N #= P*Q+Np, sp1(Np,T,R).
      sp1(0,[],[]).
```

Variables `N`, `P`, `Q`, `Np` denote finite domain variables (positive integers), variables `T`, `R`, lists of finite domain variables, `+,*` integer operations with finite domain variables; `#=/2` is the equality. Its effect is to reduce the domains of the variables in the expressions (partial arc consistency); inequality and comparison operators `#\=/2`, `#</2`, `#>/2` ... have a similar effect.

The intended meaning of the program is that `sp1` describes the scalar product relation where the first argument is an integer representing the scalar product of two vectors represented as lists (of integers).

8.2.3 Constrained Predication and *D*-Atom

A *constrained predication* is a pair (σ, a), where a is a predication and σ is a finite set of constraints. For example
`({X#=1,Y#=X+T, fd_domain(T,1,2), fd_domain(Z,2,3)}, sorted([X,Y,Z]))`
is a constrained predication in Calypso (`fd_domain/3` assigns their domains to the finite domain variables given in the first argument). For a given constraint domain D a D-atom is an object of the form $p(d_1, ..., d_n)$ where p is a predicate and $d_1, ..., d_n$ are elements of the underlying structure.

Given a set of constraints σ we denote by $Sat(\sigma)$ the set of all valuations[7] satisfying the conjunction of the constraints in σ in the interpretation of the constraint domain.

A constrained predication $a = (\sigma,\ p(t_1, ..., t_n))$, which is a syntactic object, represents the following set a_D of D-atoms:

$$a_D = \{p(\mu(t_1), .., \mu(t_n))|\mu \in\ Sat(\sigma)\}$$

[5] FD stands for Finite Domains.

[6] Calypso is the successor of `CLP(FD)` developed at INRIA-Rocquencourt and Paris 1; it is now distributed as GNU-Prolog (http://www.gnu.org/software/prolog/) [8.2]; it uses the ISO Prolog notation of clauses.

[7] A valuation is a n-uple of values in the constraint domain assigned to the corresponding elements of a n-uple of variables.

If D is the finite domain, then the example constrained predication at the beginning of this section corresponds to the following set of the D-atoms.
`{sorted([1,2,2]), sorted([1,2,3]), sorted([1,3,2]), sorted([1,3,3])}`

A constrained predication a is said to be *ground* if a_D is a singleton.

8.2.4 Operational Semantics

A CLP program is used to answer queries expressed by *goal* clauses. We assume without loss of generality that the goal has the form of a constrained predication (σ, a). The computed answer may also be seen as a constrained predication (σ', a), where σ' is called the *computed constraint*.

Example 8.2.2. For the scalar product program `Pr1` a possible goal is:

```
:- R = [_,_,_], fd_domain(R,0,20), sp1(20, [4, 3, 10], R).
```

in which case the constrained answer is:

```
({20 #= 4*X1+3*X2+10*X3, X = [X1, X2, X3],
fd_domain(X1,0,5), fd_domain(X2,0,6),fd_domain(X3,0,2)[8]},
sp1(20, [4, 3, 10], X)).
```

A state of computation can be characterised by a *resolvent* defined as a pair (σ, G) where G is a sequence of predications and basic constraints, and σ is a set of constraints, called the *constraint store*.

The initial state is determined by the goal (σ_0, g_0). The operational semantics is defined by a transition function on the states.

At each step a satisfiability test of the possibly new constraint store is performed. The satisfiability check is implementation dependent. It is usually based on a sufficient condition, so that some unsatisfiable constraint stores may pass the test. A store is said *acceptable* if it passes the satisfiability test.

The transition relation on states determines for a state (σ, G) the new state (σ', G') defined as follows:

- If G is empty the computation terminates without changing the state. The computed answer constraint is then obtained as a simplification of the formula $\exists \overline{x} \sigma$ where $\overline{x}$ denotes the free variables of σ that do not appear in the initial goal. The simplification is an implementation dependent transformation preserving the logical equivalence.
- For a non-empty G with the first element a and tail T ($G = [a|T]$) three cases are possible:
  - if a is a constraint and $\sigma \cup \{a\}$ is an acceptable store then define $\sigma' = \sigma \cup \{a\}$, and $G' = T$,
  - if a is a predication $p(\overline{t})$ then a renamed variant of a clause of the form $p(\overline{t'})$:-B is created (if any), $\sigma' = \sigma \cup \{\overline{t} = \overline{t'}\}$ and G' is obtained from G by replacement of a by B,

[8] The `fd_domain/3` constraints do not occur in the program, but are in the store as a result of values propagation. Initially `fd_domain(Xi,0,20)` holds for `i = 1,2,3`.

- if a is a constraint such that $\sigma \cup \{a\}$ is unsatisfiable or a is a predication such that there is no clause defining it then the computation halts with failure.

(σ, a) is called the *chosen (constrained) predication*.

The scalar product example shows the simplified answer constraints as constructed by Calypso.

Notice that, in the case of the Herbrand constraint domain with most general unifier used both as satisfiability check and constraint simplifier, the above described operational semantics can be seen as LD-resolution, or SLD resolution with (ISO)-Prolog computation rule [8.5].

8.2.5 CLP Search-Tree

The different computations defined by the operational semantic transition function may be represented by a tree called the *CLP search-tree* (in short search-tree) that we define as follows.

Definition 8.2.1. *CLP search-tree*
A CLP search-tree is an ordered labelled tree such that

- *The root is labelled by the initial state.*
- *Each node is labelled by a resolvent (σ, G), such that, if G is not empty, the first element of G is a predication a formed with a non constraint predicate.*
- *There are two kinds of leaf-nodes:*
  - *Nodes whose goal label G is empty (or* `true`*), called* success nodes.
  - *Nodes with a nonempty goal label, but which cannot be expanded with any acceptable store, called* failure nodes.
- *A non-leaf node labelled (σ, G) has as many children labelled (σ', G') as there are clauses l leading to acceptable constraint store as follows.*
  If $G = [a|T]$ and l is a rule, i.e. $l:\ h$:- c_0[9], $a_1, B.$, then $\sigma' = \sigma \cup c_0$, and $G' = [a_1, B|T]$ flattened[10].
  If $G = [a, c_1, a_1|T]$ and l is a fact, i.e. $l:\ h$:- c then $\sigma' = \sigma \cup c \cup c_1$, and $G' = [a_1|T]$ flattened.
  The children are in the same order as the clauses in the database[11]*. A* left-to-right order *of the children will be assumed in a two dimensional representation of the search-tree.*

Notice that one arc in the search tree may correspond to several computation steps. In order to limit the number of nodes in the search tree, successive computation steps with store accumulation only are compacted into one single step.

There are three kinds of branches: success, failed, infinite.

[9] c_0, c_1 and c below denote a conjunction of basic constraints

[10] Informally, a "flattened list" is the list in which any element which is a list has been expanded in such a way that all its elements become, in the same order, elements of the initial list.

[11] The set of the clauses of the executed program

– *A* success branch *corresponds to a success node.*
– *A* failed branch *corresponds to a failure node.*
– *An* infinite branch *corresponds to a nonterminating computation following the Prolog computation rule.*

If there is no infinite branch in a search-tree, it is a finite search-tree.

At every node the label store corresponds to the current store. To every success branch there corresponds an answer constraint *which is the current store labelling the (success) leaf.*

Notice that the notion of search-tree depends only on the predication-choice[12] (which is imposed here by the "Prolog computation rule"). The clause-choice (not specified yet) specifies the way it is visited. Given a search-tree, the execution of a goal in the context of a database, with the Prolog computation rule, may be represented by a depth-first left-to-right visit of the search-tree. This visit defines the visit order of the nodes of the search-tree, hence the order of execution of goals and subgoals. It is a total order (see below). If the search-tree has infinite branches, there is no way to visit beyond the first encountered one, which will be explored indefinitely. This explains why the execution does not terminate when the traversal visits an infinite branch. It also explains why not all solutions may be computed if there is an infinite branch with some success branches afterwards.

An example of search-tree is depicted in Figure 8.1 for the program `Pr2`

```
Pr2 c21: sp2(N,[X|S],[P|R])   :- N #= Np+P, X #= Xp+1,
                                 sp2(Np,[Xp|S],[P|R]).

    c22: sp2(N, [X|S], [P|R]) :- sp2(N, S, R).

    c23: sp2(0,[],[]).
```

and the goal

```
({N=10, P=[A,B], Q=[5,2]}, sp2(N, P, Q))
```

In the scheme shown in Figure 8.1, each non leaf node is labelled with the first predication in the resolvent and each arc is labelled with the clause used to produce it and the set of constraints added to the store in this step (the local store).

Arcs in a search-tree correspond to a computation step formalised by the use of a clause instance where the head is the chosen predication of the parent node and the first predication in the body (if any) is the chosen predication in the child node. The constraints added to the store in this step are all the constraints before the first predication in the body. In a step where some fact is chosen (its body is a conjunction of constraints), then the constraints

[12] "Predication-choice" denotes the order in which the predications of the body are chosen in a clause. The left to right order corresponds to the (standard) Prolog computation rule.

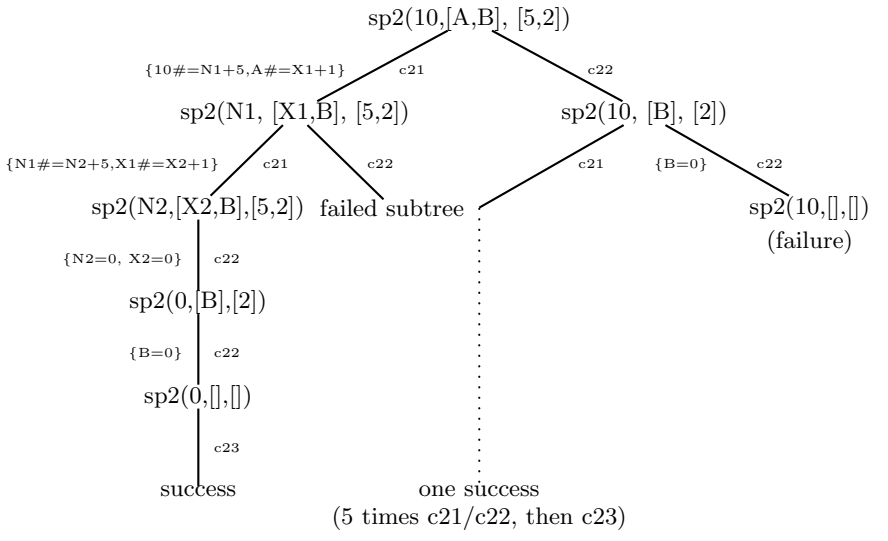


Fig. 8.1. Search-tree for the program `Pr2` and the goal `sp2(10,[A,B], [5,2])`

added to the store in this step are all the constraints of the fact with the constraints in the parent resolvent after this fact and before the following predication in the resolvent (if there is none, the parent resolvent of the parent is considered... etc).

It is important to observe that the tree traversal induced by the standard Prolog computation rule defines a total order on the nodes of the search-tree. If the nodes are coded using a Dewey-like notation (root is **1**, successive children are **11**, **12**,... and so on). The total order corresponds to lexicographic ordering of the nodes. This order will be denoted *family ordering*. It consists also of two partial orders (referred also as "family orders": the "child" relation[13] (a parent, or ancestor node is the prefix of a child or a descendent) and the "brother" relation (ordering the nodes with a common parent). If two nodes are not "descendent" neither "ancestors", we will say they are "grand cousin". The brother relation follows the order in which the clauses have been chosen to build the brother nodes. The family orders extend trivially to arcs. Figure 8.2 illustrates the relations: node **112** is descendent of node **11**, but node **122** (resp.**12**) is grand cousin of node **11** (resp. **112**).

8.2.6 Labelling

CLP systems over finite domains use "labelling" procedures to enumerate solutions when the current store is not sufficient to fix all finite domain va-

[13] Transitive closure of "child" relation is called "descendent", opposite relation is "parent", its transitive closure is "ancestor".

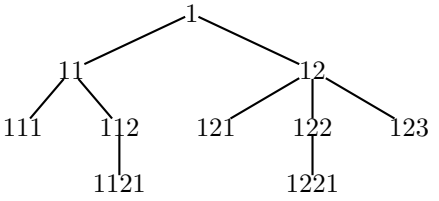


Fig. 8.2. Example CST with nodes total order (family order)

riables. Labelling procedures, with different strategies are part of CLP processors and the user may call them with a "labelling" system predicate. The user can also define its own labelling. A labelling may be visualised also as search-tree. In this case all the successful branches have the same length, corresponding to the number of finite domain variables to be enumerated.

Labelling is a common reason of poor performance and therefore focused visualisation of labelling is of great importance. It is thus particularly useful to be able to extract from the search tree the labelling part. If labelling is defined by a procedure provided by the user it will be reflected in every computation by a part of the search tree. Our specification language defined in Section 8.4 will make it possible to reduce the search tree to its labelling part.

8.2.7 State Properties

A particular view of CLP choice-tree will be specified by selecting subsets of nodes. Every node corresponds to a computation state. Thus a selected node will be identified by a property of the corresponding computation state. In practice we will restrict the property to be a property of some constrained predication. Such a property is a relation holding between the arguments of the constrained predication. We assume here that the properties are expressed with constraints (in the same CLP language)[14].

To verify that the constrained predication $(\sigma, a(X))$ satisfies the property $p(X)$ associated with a, is to prove

$D \models \sigma \rightarrow p(X)$,

hence to prove that $\sigma \wedge \neg p(X)$ is unsatisfiable.

For practical ($\neg p(X)$ cannot be easily expressed) and efficiency reasons, its proof is implemented just by trying to execute the goal $p(X)$ in the context of the current store σ. Two cases are thus possible

- The goal $p(X)$ fails. This shows that $\sigma \wedge p(X)$ is unsatisfiable and therefore that
  $D \models \sigma \rightarrow \neg p(X)$,
  the property is false.

[14] Properties can be expressed in a different language, but in this case the constraint domain D must be extended to include the interpretation of this language.

– The goal $p(X)$ succeeds. Even if the constraint solver is complete[15], it does not necessarily show that the property holds, but only that $D \models \exists (\sigma \wedge p(X))$.
  In this case the test applied does not give a decisive answer. The property may or may not hold and we decide to select the node anyway. The results will be an over approximation of the view (it will be in some sense "less" pruned than expected).

If, instead of testing directly the property $p(X)$, the user is able to express its negative counterpart such that $np(X) \Leftrightarrow \neg p(X)$ the same procedure applied with $np(X)$ instead of $p(X)$ shows, in case of failure, that $p(X)$ is true, but in case of success $\neg p(X)$ will be assumed to hold, with the risk of under approximation (too many nodes are assumed not to verify $p(X)$, hence the tree is more pruned than expected)[16].

Here is a small example. Assume the current store is σ `:{X#>=Y,X#\=Y}` and the property p(`X`,`Y`): `X#<=Y` (it does not hold). A finite domain (incomplete) solver may find that `{X#>=Y,X#\=Y,X#<=Y}` is satisfiable, hence the property is true, even if it is obviously not implied by the store.

Now assume the current store is the same but the property p(`X`,`Y`): `X#>Y` (it holds obviously). One may like to use its negation expressed as `X#<=Y`. The same solver will find that the negation holds and eliminate corresponding nodes.

In many cases however the truth of the properties will be correctly computed.

If the store consists of

```
fd_domain([X,Y,Z],1,3),Y#=X+1,Z#=Y+1
```

and the property is `integer(X)`,

the solver succeeds on the store

```
fd_domain([X,Y,Z],1,3),Y#=X+1,Z#=Y+1,integer(X).
```

which is a way to test whether a finite domain variable is "ground".

Term properties may also be tested that way, testing for example that the predicate name of a predication `Term` is `foo/1` can be expressed with the property: `p(Term) :- functor(Term,foo,1)`

If the solver is complete, the implication problem becomes decidable for some properties expressed with basic constraints. If the property p may have a negative counterpart np expressed in the same language such that always one of the property is true, when both cannot be true together, it is thus sufficient to test unsatisfiability of $\sigma \wedge p$ or $\sigma \wedge np$ (one of the formula is) to know whether p or np holds[17].

[15] A solver is complete if a success implies the existence of a solution which satisfies the answer constraint store.

[16] The proof of $D \models \sigma \rightarrow p(X)$ can thus be implemented using the negation by failure with the goal $(\sigma,\backslash{+}np(X))$, with the risk of over approximation.

[17] If $\sigma \wedge p$ is unsatisfiable, np holds, hence $\neg p$, otherwise if $\sigma \wedge np$ is unsatisfiable, p holds. See [8.16] or Chapter 3 for assertion checkers with complete solvers.

8.3 Tree Pruning and CLP Choice-Tree

CLP search-tree views may be obtained after several successive pruning operations. A "pruning" is viewed here as an operation which takes an initial tree and produces a *unique minimal* one in which the family orders are preserved. "Minimality" means that a minimal number of nodes is kept. Uniqueness means that for a given selection of nodes and arcs the "minimal" tree is unique. Such property is desirable from the user point of view since views can be thus understood declaratively.

The properties studied in this section rely only on the (ordered tree) structure of the CLP search-tree. We thus focus on ordered tree pruning, and introduce the declarative notion of the unique minimal ordered tree and an algorithm to compute such a tree.

8.3.1 Unique Minimal Pruned Tree

We first consider the problem of defining a unique minimal pruned tree issued from a subset of nodes and arcs of a given tree.

An ordered tree (OT) is a tree with a family ordering. This ordering is completely defined by the lexical ordering defined on the set of nodes coded by a Dewey like notation. The *descendent* relation is defined by the prefix ordering (node uv is descendent of node u).

It is well-known that any subset of such coded nodes, including the root, defines an ordered tree such that the filiation relations are preserved between the nodes of the subset (i.e. *descendent* , *brother* or *grand cousin* nodes are still *descendent* , *brother* or *grand cousin* in the subset).

A first transformation consists thus in selecting a subset of nodes and to take this subset as the (trivially) unique minimal tree. It is *minimal* in the sense of set ordering by inclusion. Figure 8.3 illustrates how the family order is preserved in this case (selected nodes are underlined).

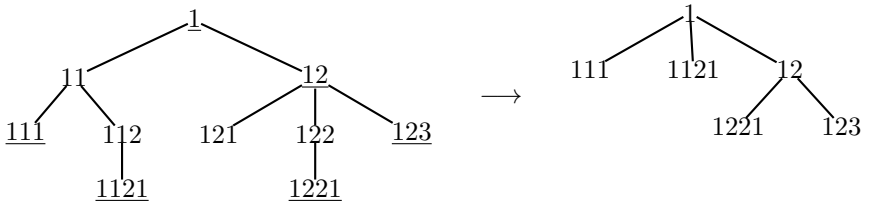


Fig. 8.3. Preservation of family ordering (selected nodes are underlined)

Selecting nodes only is not sufficient: it may be more convenient to select arcs too (or instead). The reason is that it is sometimes easier to identify

an interesting transition rather than an interesting computation state. We will see that selecting arcs (without selecting extremity nodes) results in a selection of nodes. If for example all arcs of a tree are selected, then the resulting tree is the same, i.e. all nodes are selected. Now if on a path only two non adjacent descendent arcs are selected (without their extremity nodes), one would like to see in the resulting tree two adjacent (and descendent) arcs only, in order to satisfy the *minimality* objective. The question now is which node should be selected common to both new arcs? if there are several nodes between the two initial arcs, there are several candidates.

In order to guarantee *minimality* and *uniqueness* of the result, one node only (according to the family ordering) among possible candidates will be selected.

Figure 8.4 shows a possible minimal tree when nodes and arcs are selected.

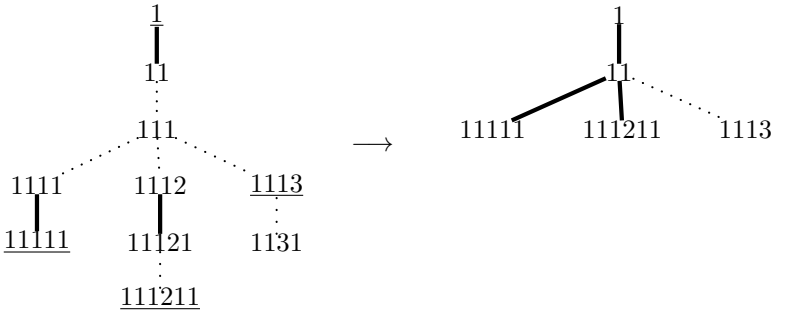


Fig. 8.4. OT pruning with arc selection (selected arcs are bold).

Notice that in Figure 8.4 non selected node **11** remains, to preserve the *grand cousin* relation, and there is an arc **(11, 1113)** which corresponds to a path in the original tree which contains non selected arc only. There is a 1-1 correspondence between the selected arcs of the initial tree and a subset of arcs of the pruned tree.

Notice also that one could have chosen node **111** instead of node **11**. The minimal ordered tree is not unique.

The following definition summarizes the ideas above and is a declarative definition of minimal ordered tree obtained from nodes and arcs selection.

Definition 8.3.1. *Pruned ordered tree*

Given an OT and a subset of selected nodes (the root of the OT is always selected) and arcs, the pruned OT is the least subset of nodes such that:

1. *all selected nodes are in the pruned OT,*
2. *to each original arc corresponds an arc in the pruned tree and there is a 1-1 correspondence between the set of original selected arcs and the set of their images in the pruned OT.*

3. the filiation orders of the nodes and the arcs are preserved,

As we have seen, the pruned tree, as defined, is not always unique. There are several possible nodes between two non adjacent descendent arcs whose extremity nodes have not been selected. Considering the fact that the family orders must be preserved, candidate nodes are always nodes of a portion of path in the initial ordered tree. The problem arises only when there is an arc whose lower node is not selected. If there are several branches issued from that node with selected elements all nodes on the section common to all paths are candidate (Nodes of path (**11,111**) in Figure 8.4). In order to make the pruned ordered tree unique, the least node (i.e. the lower node of the arc, or the upper node in the common path) will be selected[18].

If we define now an ordering on the pruned trees as follows

Definition 8.3.2. *OT Ordering*

Given an OT and two subset of nodes s1 and s2 defining respectively the OT's t1 and t2, t1 < t2 iff

s1 has less elements that s2, or

s1 and s2 have the same cardinality and for each element of s1 there is an element of s2 which is equal or greater according to the total family ordering.

Hence the definition.

Definition 8.3.3. *Minimal pruned OT*

Given an OT and a nonempty set of selected nodes (the root of the OT is always selected) and arcs, the minimal pruned OT is the least tree such that:

1. *all selected nodes are in the pruned OT,*
2. *to each original arc corresponds an arc in the pruned tree and there is a 1-1 correspondence between the set of original selected arcs and the set of their images in the pruned OT,*
3. *the filiation orders of the nodes and arcs are preserved.*

It follows from the definition and explanations above that there is a unique minimal pruned OT (e.g. in Figure 8.4, the node **11** is chosen among the candidates {**11,111**}).

8.3.2 Minimal Pruned Tree Construction

Proposition 8.3.1. *Construction of the minimal pruned OT*

Given an OT and a nonempty subset of selected nodes and arcs, there is a one-pass algorithm to compute the nodes of the minimal pruned OT.

[18] This choice is arbitrary. Any other candidate node in the common path could be chosen and condition 3 of definition 8.3.1 could be even better understood using lower node instead of the upper one. The upper node has been chosen here because it is attached to a unique selected arc (the lower node may not belong to any selected arcs or be common to several ones). We did not study the practical impact of a different choice on the pruned trees.

We assume that a subset of nodes and a subset of arcs is selected. If the lower node of a selected arc is selected too, there is no need to further action since there will be a corresponding arc in the pruned OT. So if all selected arcs have their lower node selected, there is nothing to do more than building a new tree from the subset of selected nodes.

A question arises if the lower node of an arc is not selected. Some descendent node should exist in order to ensure that there will be a corresponding arc in the pruned OT. Two cases only need to be considered

- There is one selected descendent node which is before all other selected elements (according to the total family ordering) in the subtree issued from that arc. In this case this selected node will serve as lower node for the corresponding arc in the pruned OT.
- In all other cases (for example two descendent arcs with no selected element between them or no selected elements in the subtree issued from the arc, or two selected nodes direct descendent in different paths issued from that arc, ...), the lower node of the arc will be selected.

The purpose of the following algorithm is to decide whether the lower node of a selected arc must be selected. When this is known, the (unique) minimal pruned OT is exactly the subset of selected nodes. The construction of the minimal pruned OT is thus processed according to the following phases

1. node and arc selection,
2. determination of the selected lower nodes of selected arcs,
3. construction of the pruned tree from the set of selected nodes.

Each phase needs at least one pass (complete in the worst case) over the initial OT. For the first phase the selection criterion is applied to each node (and additionally arc). In the second phase, a complete (bottom-up) pass over the subtree is needed (in the worst case) to decide whether the lower node of a selected arc is selected or not. The third phase corresponds to a re-numbering of the nodes. Its efficiency depends on the used data structure, but actions will at least follow the total ordering of the nodes (hence one more pass).

Algorithm

Each node has two flags denoted **Fa** and **Fb**. The initial values of **Fa** result from node and arc selection and may include a value "?", the values of **Fb** are thus computed and values "?" removed from values of **Fa**.

Fa $\in \{+,-,?\}$, respectively "selected", "not selected" and "to be decided" (this last value is assigned to a lower non selected node of a selected arc).

Fb $\in$ {n,ya,y,o}, respectively "nothing selected in the subtree", "one selected arc only in direct descendence or lower node of a selected arc", "one selected node only in direct descendence (possibly including this node)", "other".

The problem is to replace each "?" by "+" or "−".

Flag **Fa** is initialised in the first phase and updated in the second, flag **Fb** is also set in the second phase. Thus, assuming that the values of **Fa** have been initialised, final values can be fixed at every node according to the following test cases.

We describe the different test cases in the form: initial value of **Fa**, condition, →, final value of **Fa** (if changed), value of **Fb**.

– leaf node:
  + → y
  − → n
  ? → +, ya
– nonleaf node:
  + → y
  − and all children nodes flagged n → n
  − and one child only flagged y others flagged n → y
  − and one child only flagged ya others flagged n → ya
  − otherwise → o
  ? and one child only flagged y others flagged n → −,ya
  ? otherwise → +,ya

The point now is to observe that

1. given the values of **Fa**, the values of **Fb** can be computed in one (bottom-up) pass over the OT;
2. if node and arcs can be selected during a first pass, first two phases can be performed together in one complete pass over the initial OT; hence property 8.3.1.

It is here assumed that the value of flags **Fb** of the children of a node can be accessed with no significant cost from any nonleaf node.

8.3.3 CLP Choice-Tree

Pruning a CLP search-tree does not results in a CLP search-tree. So in order to be able to compose successive prunings we need to introduce the (conservative by pruning) notion of CLP choice-tree. For practical reasons we also consider finite trees only.

Definition 8.3.4. *CLP choice-tree*
A CLP Choice-tree is a finite ordered labelled tree such that

– *Each node is labelled by a resolvent* (σ, G),
– *Each edge issued from a non-leaf node* u *labelled* (σ, G) *and with extremity node* v *labelled* (σ', G') *is labelled by a "local" store* μ *such that* $\sigma' = \sigma \cup \mu$.
– *There are three kinds of leaf-nodes:*
  – success*: some continuation of computation leads to a success, or* G *is empty (existence of a successful computation).*

  - failure*: any attempt to continue leads to rejection or* failure *(failed computation or absence of successful computation).*
  - nonterminated *otherwise (stopped long computations for example).*
- *Each node u is marked*
  - success *if u is root of a success branch in the initial choice-tree*[19].
  - failure *if u is root of a finitely failed subchoice-tree in the initial choice-tree.*
  - nonterminated *otherwise.*
- *All nodes are totally ordered by a* family ordering .

According to definition 8.3.4, any finite CLP search-tree can be viewed as a CLP choice-tree. Labels of the nodes are resolvents, labels of the arcs are stores. Success (resp. failed) leaves are marked success (resp. failure). All nodes are totally ordered by the visit order. The definition 8.3.4 shows that CLP choice-trees a preserved by the pruning operation. This allows for composition of prunings. The proposition 8.3.2 shows some properties preserved by pruning.

Proposition 8.3.2. *Composition of prunings*

Any CLP choice-tree obtained after a finite number of prunings satisfies the properties:

- *All the nodes are nodes of the initial tree and they keep their initial status (success, failure or nonterminated).*
- *Every arc corresponds to a path in the initial tree and is labelled by the union of local stores of this path.*
- *Every node marked "success" correspnd to a possibly successful computation.*
- *Every node marked "failure" corresponds to a node whose subchoice-tree is finitely failed.*
- *Every node is marked "nonterminated" otherwise.*

8.4 A Language to Specify Views

We finally turn to the question how to specify nodes and arcs selection. We present first a non exhaustive collection of criteria whose combinations allow to define useful views[20]. It is a combination of predefined general properties of choice-tree nodes (like "to be a leaf") and of program properties. These are properties of the chosen predication of the resolvent, which is referred by one argument of the predefined specification predicate (usually the first one). But the chosen predications always correspond to (instance of) predication in the program.

[19] Remark that a non-leaf success node may be the root of a failed reduced subchoice-tree.

[20] The question whether any view can be specified with the proposed collection of criteria is not considered here.

8.4.1 Node Selection

Can be selected all nodes such that

- *A property of a predication*: The chosen predication a of the resolvent satisfies the given property: if σ is the current store it is a property of the constrained predication (σ, a). Example: a given argument is of type "finite domain or list of finite domain variables".
  It is expressed by `node_property(n,l)`, where `n` is a choice-tree node and `l` a list of properties, understood as their disjunction. A property is defined by the user as a unary predicate whose single argument is the chosen predication of the resolvent of node `n`.
- *success leaf*: to be a success leaf, expressed by `success_leaf(n)`.
- *failure leaf*: to be a failure leaf, expressed by `failure_leaf(n)`.
- *success node*: to be a node, root of a success branch, expressed by `success_node(n)`. It includes the success leaves.
- *failure node*: to be root of a failed subchoice-tree, expressed by `failure_node(n)`. It includes the failure leaves.
- *leaf*: to be a leaf, expressed by `leaf(n)`.
- *least failed tree root*: to be root of a "biggest" failed subtree, expressed by `least_failed_tree_root(n)`.
- *node of some selected arc*: there is the possibility to specify which nodes are selected with a selected arc: (*no node, upper, lower node* or *both nodes*). The way to express it is with arc selection.

8.4.2 Arc Selection

Can be selected all arcs such that

- *A property of the chosen predication of the lower node*: the chosen predication of the resolvent of the lower node satisfies the property.
  It is expressed by `in_arc(n,l,s)`, where `n` is a choice-tree node, `l` a list of properties, and `s` $\in$ { `none`,`upper`,`lower`,`both` }, called *node selector*, (`none`,`upper`,`lower`,`both` stand respectively for "no node", "upper", "lower" and "both nodes").
- *A property of the chosen predication a of the upper node with local constraints*: if the current store of the lower node is σ, the chosen predication a is such that the constrained predication (σ, a) satisfies the given property.
  It is expressed by `out_arc(n,l,s)`, where `n` is a choice-tree node, `l` a list of properties of the parent node, and `s` the node selector.
- *A label property*: the label of the arc satisfies some property (the store includes some constraint or the local store has some property).
  It is expressed by `label_property(l,s)`, where `l` a list of properties, and `s` the node selector. Properties depends on the nature of the labels.

– *A program arc*: a clause and a body (non constraint) predication number (first predication corresponds to the arc $h...a_1$, others are $a_i...a_{i+1}$ if a_i is defined by facts only).
It is expressed by `clause_arc(c,i1,i2,l,k,s)`, where `c` denotes a clause of the user's program, `i1` and `i2` are two successive integers between 0 and the number of predications in the body of the clause, `l` is a list of properties, `k` denotes the node to which this property applies (`k` ∈ {`u`,`l`}), and `s` the node selector.

List of properties are interpreted as their disjunction.

8.4.3 Node and Arc Selection Using Differential Information

The selection criteria above use properties holding at a single node. As they are properties of one computation state, there is no way to take into account more global information (i.e. information concerning several successive states) without modifying the program in an "ad-hoc" manner[21]. For example, it would not be possible to express at a single state that the cardinality of the domain of a FD variable has been significantly modified (for example divided by 2). Our intention is to avoid as much as possible to have to modify the program. We therefore extend the selection language in order to allow selection by some "differential property", i.e. using different properties holding at different nodes.

For implementation efficiency reasons, it is not possible to verify properties in a given criterion with different stores (only the current store can be used at some node). It is thus necessary to "record" some information about properties holding with the current store. It is the purpose of the "store property" instruction which is introduced now for node and arc selection based on differential properties. For such purpose, it is also needed to introduce general properties as binary relations which relate predications with some other domain. If the other domain is *bool* such relation is a way to express that a given property holds or does not.

– *store property*: some information is stored at some node if the chosen predication of the resolvent satisfies the property.
It is expressed by `set_prop(n,l)`, where `n` is a choice-tree node, `l` a list of binary properties. If the current store of node `n` is σ, the chosen predication is a and p is one of the binary properties with second argument t, then if $(\sigma, (a, t))$ satifies p, the pair consisting of (`p/2`,t) is recorded at node `n`. If t has variables, a renamed copy of t is recorded.

[21] In fact two successive computation states are related by a path in the search tree and a succession of "calls" of different predicates (or the same in the case of a recursive one) in the program. If one want to relate two states in a property of a single sate, some information must be transmitted in the form of an extra argument and likely some extra computation.

– *node ancestor*: a node is selected if some ancestor verifies some property.
  It is expressed by `ancestor(n,p,q)` where `n` is a choice-tree node, `p` and `q` two binary properties. If there exists at some ancestor node `n'` a pair (`p/2`, t), and the current store of node `n` is σ and the chosen predication is a, and $(\sigma, (a, t))$ satifies q, then the node `n` is selected.
  If there is no such ancestor node `n'` the criterion fails and the node `n` is not selected.
– *differential arc*: an arc is selected if lower and upper nodes are related by some relation.
  It is expressed by `diff_arc(n,p,q,s)`, where `n` is a choice-tree node, `p` and `q` two binary properties and `s` a node selector. If the parent node `n'` has a pair (`p/2`, t)., and the current store of node `n` is σ and the chosen predication is a and $(\sigma, (a, t))$ satifies q, then the arc whose lower node is `n` is selected.
  If there is no such pair at node `n'` the criterion fails and the arc is not selected.

8.4.4 View Specification

Finally a pruning is specified by the union of two criteria selecting respectively nodes and arcs (with possibly selected nodes). For node selection union, intersection and complementation of criteria are allowed. For arcs the union is allowed only.

Prunings may be composed (i.e. a pruned tree may need some further pruning). This is needed since for example arc selection may re-introduce nodes which have not been explicitly selected. For example if one wants to select arcs in the subchoice-tree limited to the success nodes only, the criterion `success_node(N) union out_arc(N,[prop],both)` with the property `prop` selecting a specific subset of nodes (for example by the name of the predication), may select some failure nodes corresponding to lower nodes of selected arcs. Thus the way to obtain success nodes only in the final pruning is to compose both criteria.

8.5 Implementation and Examples

We started an implementation of CLP choice-tree visualization and pruning for Calypso.

The initial CLP choice-tree corresponds to the CLP search-tree as described in Section 8.2.5. We call it the "concrete choice-tree" (CCT), from which all prunings will be computed. This CCT may not be finite or just too big to be tractable. Therefore we need some "termination" criteria. Such criteria are also a way to impose some limitation on the size of the CCT in case of non terminating or too long computation.

The following termination criteria may be used[22]:

- Maximal depth (default value: unlimited),
- Number of solutions (default value: unlimited),
- Number of failure nodes (default value: unlimited),
- Number of nodes (default value: unlimited),
- Number of leaves (default value: unlimited),
- Execution time (default value: unlimited),
- Resource exhausted,
- User interrupt.

In order to be able to deal with large CCT, only some information is stored at each node (basically the chosen clause, the chosen predication and the "set" properties when requested). To obtain any other information at some node (for example the current store) needs recomputation along the path to this node. Node and arc selection is performed for each different pruning after the CCT has been computed once.

Different views can be thus derived from the same CCT, which may not be displayed.

8.5.1 Displaying the Full Concrete Choice-Tree

Figure 8.5 shows a concrete choice tree for the following program and the goal `goat(L)`.

```
goat(L) :-  go1(L).
goat(L) :-  go2(L).

go1(L)  :-  L = [X1, X2, X3], fd_domain(L, 1, 5),
                              user_labeling(L,1,L), X1 + X3 #<= X2.

go2(L)  :-  L = [X1, X2, X3], fd_domain(L, 1, 5),
                              X1 + X3 #<= X2, user_labeling(L,1,L).

user_labeling([],_,_).
user_labeling([X | L],N,L0) :- fd_min(X, Min), X #= Min,
                               N1 is N+1, user_labeling(L, N1,L0).
user_labeling([X | L],N,L0) :- fd_min(X, Min), X #> Min,
                               user_labeling([X | L], N,L0).
```

This program demonstrates two (elementary) ways of posting finite domain constraints and using an user defined labelling. After a serie of deterministic decuctions (path from the root), there are two subchoice-trees. In the first (corresponding to the goal `go1(L)`) the labelling is performed on the unconstrained variable list and the constraint is posted at the end leading to success or failure. This case corresponds to the worst combinatorial complexity (125 branches). In the second case (`go2(L)`) the constraint is posted before the labelling and the number of cases is restricted (20 branches).

[22] Not implemented yet.

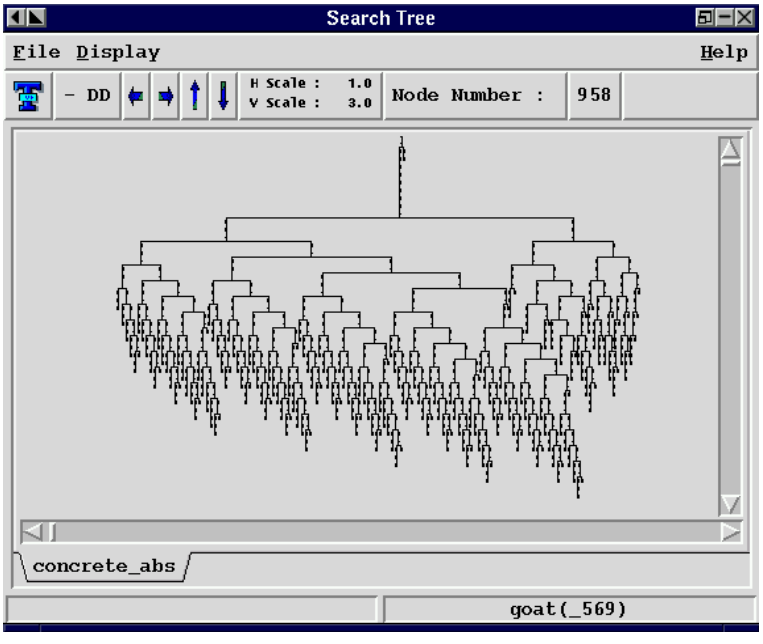


Fig. 8.5. Display of a full CCT

Displaying the CCT is not the best way to understand what is happening (here 958 nodes are depicted!). It just reveal the recursive nature of the program. This predefined ("identity") pruning is called here `concrete_abs`.

8.5.2 A General Criterion for a More Synthetic View

A way to obtain a more syntetic view is to restrict the display to success nodes corresponding to execution of user defined predicates only and to collapse failed subchoice-trees into a single node (its root). This can be expressed by defining a general criterion, which can be used with any Calypso program, as follows.

```
:-new_view([collapsed(collaps)]).

collaps(X)   :- least_failed_tree_root(X) union
                (success_node(X) inter node_property(X,[udf_prop])).

udf_prop(X)  :- functor(X,F,N),atom_chars(F,[C|_]),C \= '$',
                findall(Y,predicate_property(F/N,Y),L),
                  (L=[];
                      (\+ member(native_code,L),
                       \+ member(built_in,L))
                   ).
```

The first directive `new_view` declares `collapsed` as a new view name. It has one argument which is the name of a node selection criterion: `collaps`. It is the name of a unary predicate whose argument `X` is assumed to be instantiated by the chosen predication. It is defined by a set expression using "union" and "inter" (intersection) operators, negation may also be used. `udf_prop` is a unary property which is true iff the principal functor of `X` is a non Calypso system predicate (i.e. a used defined predicate). Figure 8.6 shows the corresponding view for the same program which has now 163 nodes only.

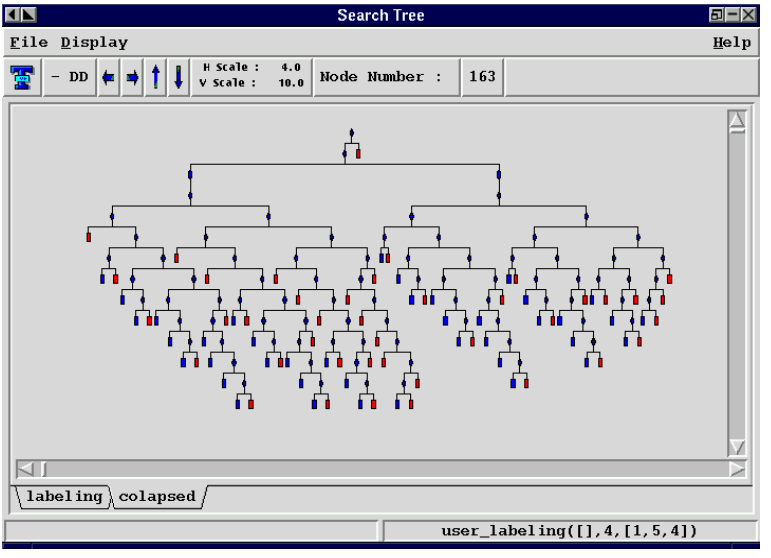


Fig. 8.6. Displaying user defined success nodes only and roots of failed subchoice-trees

8.5.3 View of the Labelling

This section shows how our selection primitives can be used for the example program to restrict visualization of its computations to the labelling phase.

This can be done with the following criterion:

```
:-new_view([labeling(go_goals,ulabeling) ]).

go_goals(X)     :- node_property(X, [prop_goals]).
prop_goals(X)   :- functor(X,NP,_), atom_chars(NP,[C1,C2|_]),
                                          C1 = 'g', C2 = 'o'.

ulabeling(X)   :- out_arc(X, [fixvar], none).
fixvar(Term)   :- Term =.. ['fd_min', X , Min], integer(X), X=Min.
```

The directive defines now a new criterion named `labeling` with two arguments: the first one is a node selection criterion name and the second an arc selection criterion. The view will result then from the selection of both nodes and arcs.

The selection of nodes is just to keep roots corresponding to the `go_goals`, i.e. nodes where predications whose principal functor name starts by "go" are executed. This preserves the display of two separate choice-trees for each initial goal (`go1,go2`).

The selection of arcs, called `labeling`, uses an `out_arc` selection criterion with property `fixvar`. This property selects those steps where the value of the labelled (finite domain) variable is "fixed" to some integer (it becomes "ground"). Indeed if one restricts observation to these steps (each finite domain variable is fixed exactly once for each of its possible values), a labelling tree as described in Section 8.2.6 will be obtained. That is to say each level will correspond to the enumeration of the different values of the same variable. In fact the property selects the first step in the second clause of `user_labeling` and all the steps for the same variable are "grand cousin". They will be displayed as children of a common parent.

The resulting view is displayed in Figure 8.7. The differences in complexity of the two subprograms are clearly depicted now. Moreover, it can be observed on the colored picture that the second tree has only success nodes.

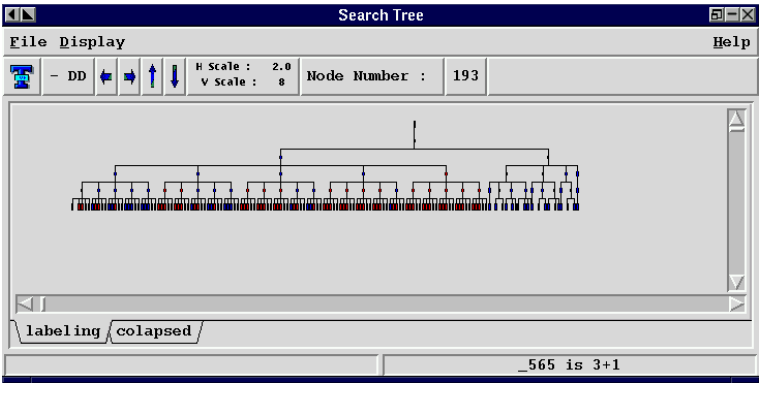


Fig. 8.7. A view which clearly shows the labelling and combinatorial complexity differences

Notice that the same view could have been obtained by introducing in the program in the right place (i.e. after `X #= Min`), an arbitrary predication and by selecting only the nodes where this predication is executed. This may be a good trick if one wants to display some related information. Here however, we wanted to obtain this view without modifying the program.

8.5.4 Use of State Properties

We illustrate on a simple example the use of state properties expressed by constraints.

```
go_list(N) :- length(L,N), fd_domain(L,1,100),
                                                order(L), mydisplay(L).
order([]).
order([_]).
order([X,Y|L]) :- X#<=Y, X#\=Y, order([Y|L]).

mydisplay([]).
mydisplay([_|L]) :- mydisplay(L).
```

It defines a list of `N` finite domain variables (`length/2`) with values ranging over `[1..100]` (`fd_domain/3`) and posts ordering constraints (`order/1`) in such a way that possible values in the resulting list are in strict increasing order. `mydisplay/1` creates a deterministic linear branch in the choice-tree corresponding to its recursive execution with one argument which is a tail of the "ordered" list.

The objective is now to show a view with this branch only when the list has at least two ordered variables (i.e. such that `X < Y`). As we have seen in Section 8.2.7 it is possible to use the negation of the property, i.e. `X #>= Y`. This is expressed by the following new criterion:

```
:-new_view([prop1(apply_nd1)]).

apply_nd1(T)  :- T =.. [mydisplay,[X,Y|_]], \+propr1([X,Y]).
propr1([X,Y]) :- X#>=Y.
```

Unfortunately, due to the incompleteness of the finite domain calypso solver `propr` always succeeds, hence `\+propr([X,Y])` always fails and no node is selected: an under-approximation is obtained (no node is displayed in the view).

If one uses instead the following criterion:

```
:-new_view([prop2(apply_nd2)]).

apply_nd2(T)  :- T =.. [mydisplay,[X,Y|_]], \+propr2([X,Y]).
propr2([X,Y]) :- X#>=Y, fd_labeling([X,Y]).
```

Using the labelling to prove the property makes the solver complete and the right result is obtained as depicted in Figure 8.8 with the goal `go_list(10)`.

8.5.5 Comparison of Labelling Strategies

We terminate this chapter with a crypt-arithmetic example showing the different labelling search with 2 different strategies. Here is the problem: to find all the (different) digits satisfying the equation

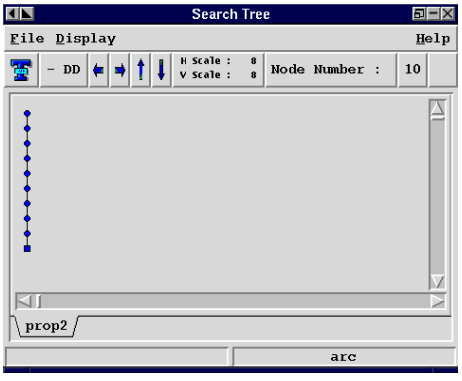


Fig. 8.8. Node selection with state property and "complete" solver

```
  B A I J J A J I I A H F C F E B B J E A
+ D H F G A B C D I D B I F F A G F E J E
-----------------------------------------
= G J E G A C D D H F A F J B F I H E E F
```

Here is a possible program:

```
crypta(LD,Stra):-
    fd_set_vector_max(9),
    LD=[A,B,C,D,E,F,G,H,I,J],
    fd_domain(LD,0,9),
    fd_domain([Sr1,Sr2],0,1),
    fd_domain([B,D,G],1,9),
    fd_all_different(LD),

       A+10*E+100*J+1000*B+10000*B+100000*E+1000000*F+
       E+10*J+100*E+1000*F+10000*G+100000*A+1000000*F
    #= F+10*E+100*E+1000*H+10000*I+100000*F+1000000*B+10000000*Sr1,

       C+10*F+100*H+1000*A+10000*I+100000*I+1000000*J+
       F+10*I+100*B+1000*D+10000*I+100000*D+1000000*C+Sr1
    #= J+10*F+100*A+1000*F+10000*H+100000*D+1000000*D+10000000*Sr2,

       A+10*J+100*J+1000*I+10000*A+100000*B+
       B+10*A+100*G+1000*F+10000*H+100000*D+Sr2
    #= C+10*A+100*G+1000*E+10000*J+100000*G,

    fd_labelling(LD,Stra).
```

The view of the system labelling can be specified in Calypso as follows:

```
:-new_view([systemlab(_,syslabeling)]).

syslabeling(X) :- in_arc(X, [sys_lab],lower).

sys_lab(Term):- Term =.. ['$fd_labeling_mth',A,B,C,D];
                Term =.. ['$fd_labeling_std',A,B].
```

There is one solution only, hence one success branch. All other branches are failed branches.

The Figure 8.9 shows the search with the labelling strategy "max-regret" in which (in short) the first labelled variable has the largest subdomain: there is a great number of failed branches.

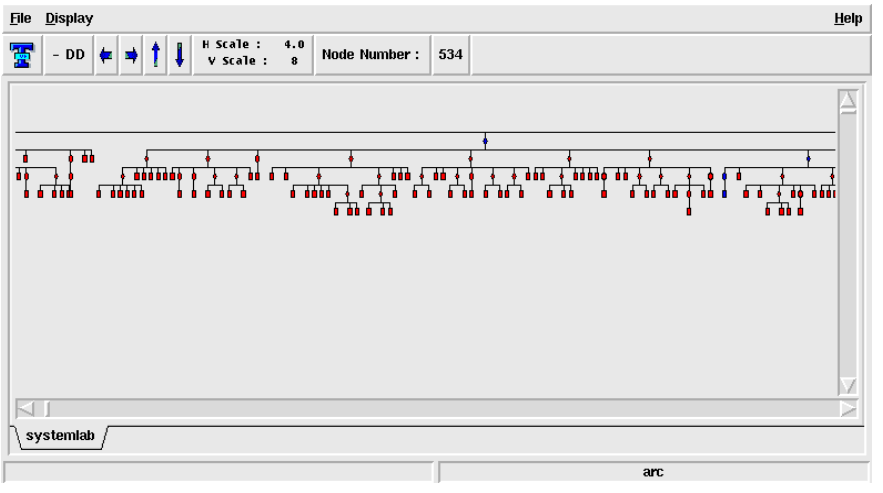


Fig. 8.9. Crypta with "max-regret" labelling strategy

In the Figure 8.10 the "most-constrained" labelling strategy is used: the variable with the smallest domain is labelled first[23].

It can be observed from these views how domains are reduced more quickly with the second strategy and how the number of failure nodes becomes limited.

8.6 Conclusion

The ideas presented here are part of a prototype debugging environment for Calypso. This environment includes many other functionalities, not presented here, as facilities to display a view, node informations, constraints store and finite domain variable evolution, etc, ... and is connected with some other debugging tools described in this book (static type analysis of Chapter 4 and declarative debugging of Chapter 5). Neither visualization of constraint store nor constraint propagation have been considered here. A constraint debugging

[23] If two variables have the same domain cardinality, the one which is used in more constraints is chosen.

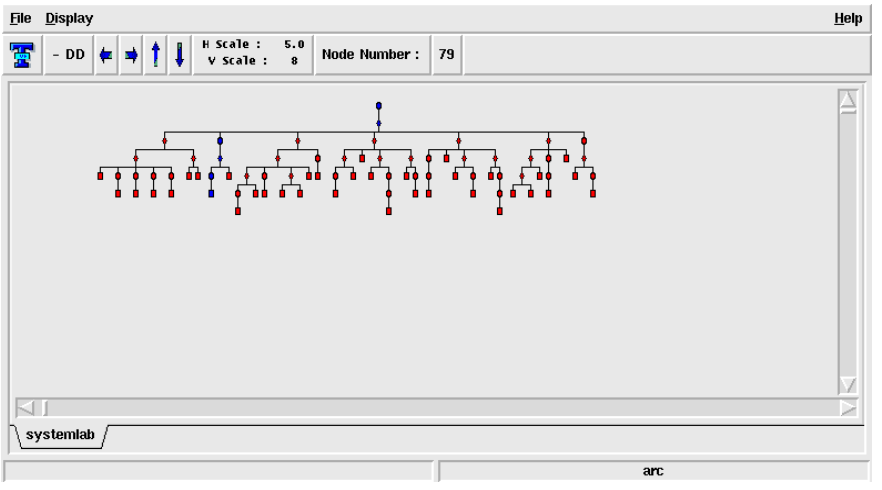


Fig. 8.10. Crypta with "most-constrained" labelling strategy

tool, based on visualization, should include such modules, like those described in Chapter 10 and Chapter 11.

We focused here on the computation flow and control visualization starting from search-trees, formalized as SLD-trees in logic programming. The visualizers presented in other chapters of this book are based on a slightly different models (Byrd's box model in Chapter 6, AND/OR trees in Chapter 9). In Chapter 7 the focus is on labelling only.

The originality of our approach lies in its genericity (several search space views may be defined on the fly for the same program and run) and the way to specify the views, in particular, using program predicate properties. The specification language is extremely simple and the user defined properties are expressed in the same CLP language. Views may be understood as "bias" which allow the user to identify some remarkable properties (redundancies, symmetries, ...).

The presented tools are still under development and experimentation. We presented here some new research direction. The right pruning specification language can only be a compromise between expressiveness and efficiency and also between what is simpler: modifying the program or writing a criteria. It may also depend from applications or constraint domains.

Choice-tree pruning could also be considered from a different point of view: choice-tree of an abstracted program. This could be also an interesting approach. In fact some properties may be captured by an abstract program (like predicate types for example), or an abstract program may terminate when the original one does not, and allow to study some properties. We did not choose this approach, since our purpose was to find ways to study

concrete choice-trees, and the definition of abstract domains for performance debugging needs further investigations.

The use of predicate properties needs some efficient method for verification, as the number of nodes which must be tested may be enormous. If properties are expressed with constraints this leads to the entailment problem. Meta constraints in Calypso allow to solve this problem efficiently in some cases [8.8]. This introduces restrictions on the use of constraints in properties. We preferred to test directly properties with the risk to obtain over (or even under) approximations of prunings. First examples show the relevance of this approach with regards to efficiency and interest of the approach.

References

8.1 Cosytec. *CHIP user Manual, beta release.* Sept. 1998.

8.2 D. Diaz. *GNU-Prolog user Manual.* http://pauillac.inria.fr/ diaz/gnu-prolog/

8.3 The DiSCiPl Consortium. Esprit LTR (Task 4.2) Project Nb 22532 *Debugging Systems for Constraint Programming.* http://discipl.inria.fr, October 1997-June 1999.

8.4 The DiSCiPl Consortium, ed. P. Deransart. *CP Debugging: Tools.* http://discipl.inria.fr/deliverables1.html, D.WP1.1.M1.1-2, July 1997.

8.5 P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual.* Springer Verlag, 1996.

8.6 P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

8.7 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

8.8 B. Carlson, M. Carlsson, and D. Diaz. *Entailment of Finite Domain Constraint.* ftp://ftp.inria.fr/INRIA/Projects/loco/publications/entail.ps

8.9 M. Meier. *Debugging Constraint Programs.* Proceedings of Principles and Practice of Constraint Programming (CP95), Cassis, pp 204–221, 1995.

8.10 J. Jaffar and M.J. Maher. *Constraint Logic Programming: A Survey. Journal of Logic Programming*, 19/20:503–581, 1994.

8.11 PrologIA. *Prolog IV Manual, Tutorial, Basic Concepts.* 1997.

8.12 F. Fages. *Programmation Logique par Contraintes.* Ellipses, X-Ecole Polytechnique, France, 1996.

8.13 J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

8.14 C. Schulte. *Oz Explorer: A Visual Constraint Programming Tool.* Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97), Leuven, The MIT Press, pp 286–300, July 1997.

8.15 C. Schulte. *Programming Constraint Inference Engine.* Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (ICLP'97), Linz, Springer-Verlag, LNCS 1330, pp 519–534, October 1997.

8.16 K. Marriott and P. Stuckey. Programming with constraints, an introduction. The MIT Press, 1998.

9. Tools for Search-Tree Visualisation: The APT Tool

Manuel Carro and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, S-28660-Madrid, Spain
email: {mcarro,herme}@fi.upm.es

The control part of the execution of a constraint logic program can be conceptually shown as a search-tree, where nodes correspond to calls, and whose branches represent conjunctions and disjunctions. This tree represents the search space traversed by the program, and has also a direct relationship with the amount of work performed by the program. The nodes of the tree can be used to display information regarding the state and origin of instantiation of the variables involved in each call. This depiction can also be used for the enumeration process. These are the features implemented in APT, a tool which runs constraint logic programs while depicting a (modified) search-tree, keeping at the same time information about the state of the variables at every moment in the execution. This information can be used to replay the execution at will, both forwards and backwards in time. These views can be abstracted when the size of the execution requires it. The search-tree view is used as a framework onto which constraint-level visualisations (such as those presented in the following chapter) can be attached.

9.1 Introduction

Visualisation of CLP executions is receiving much attention recently, since it appears that classical visualisations are often too dependent on the programming paradigms they were devised for, and do not adapt well to the nature of the computations performed by CLP programs. Also, the needs of CLP programmers are quite different [9.14]. Basic applications of visualisation in the context of CLP, as well as Logic Programming (LP), include:

- Debugging. In this case it is often crucial that the programmer obtain a clear view of the program *state* (including, if possible, the program point) from the picture displayed. In this application, visualisation is clearly complementary to other methods such as assertions [9.2, 9.10, 9.5] or text-based debugging [9.6, 9.12, 9.15]). In fact, many proposed visualisations designed for debugging purposes can be seen as a graphical front-end to text-based debuggers [9.11].
- Tuning and optimising programs and programming systems (which may be termed—and we will refer to it with this name—as *performance debugging*). This is an application where visualisation can have a major impact,

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 237–252, 2000.

possibly in combination with other well-established methods as, for example, profiling statistics.
– Teaching and education. Some applications to this end have already been developed and tested, using different approaches (see, for example, [9.13, 9.16]).

In all of the above situations, a good pictorial representation is fundamental for achieving a useful visualisation. Thus, it is important to devise representations that are well suited to the characteristics of CLP data and control. In addition, a recurring problem in the graphical representations of even medium-sized executions is the huge amount of information that is usually available to represent. To cope successfully with these undoubtedly relevant cases, *abstractions* of the representations are also needed. Ideally, such abstractions should show the most interesting characteristics (according to the particular objectives of the visualisation process, which may be different in each case), without cluttering the display with unneeded details.

The aim of the visualisation paradigms we discuss is quite broad—i.e., we are not committing exclusively to teaching, or to debugging—, but our focus is debugging for correctness and, mainly, for performance. Visualisation paradigms can be divided into three categories (which can coexist together seamlessly, and even be used together to achieve a better visualisation): visualising the execution flow / control of the program, visualising the actual variables (i.e., representing their run-time values), and visualising constraints among variables. The three views are amenable to abstraction.

In this chapter we will focus on the task of visualising the execution flow of constraint logic programs. This is complementary to the discussion on depiction of items of data and their relationship (e.g., the constraints themselves), that will be discussed in Chapter 10. Herein, we will discuss general ways in which the control aspects of CLP execution can be visualised. Also, we will briefly describe APT, a prototype visualiser developed at UPM which implements some of the ideas presented in the chapter.

9.2 Visualising Control

Program flow visualisation (using flowcharts and block diagrams, for example) has been one of the classical targets of visualisation in Computer Science. See, for example, [9.3] or the multiple versions of animated sorting algorithms (e.g., those under `http://reality.sgi.com/austern/java/demo/demo.html` or `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`), for some depictions of data evolution in procedural and object-oriented paradigms. The examples shown there have a strong educational component, but at the same time can be used for studying performance problems in the algorithms under consideration.

One of the main characteristics of declarative programming is the absence of explicit control. Although this theoretical property results in many advantages regarding, for example, program analysis and transformation, programs are executed in practice with fixed evaluation rules,[1] and different declaratively correct programs aimed at the same task can show wide differences in efficiency (including termination, which obviously affects total correctness). These differences are often related to the evaluation order. Understanding those evaluation rules is important in order to write efficient programs. In this context, a good visualisation of the program execution (probably combined with other tools) can help to uncover performance (or even correctness) bugs which might otherwise be very difficult to locate.

In CLP programs (especially in those using finite domains) it is possible to distinguish two execution phases from the point of view of control flow: the *programmed search* which results from the actual program steps encoded in the program clauses, and the *solver operations*, which encompass the steps performed inside the solver when adding equations or when calling the (generally built-in) enumeration predicates. These two phases can be freely interleaved during the execution.

9.3 The Programmed Search as a Search Tree

The programmed search part of CLP execution is similar in many ways to that of LP execution. The visualisation of this part of (C)LP program execution traditionally takes the form of a direct representation of the search-tree, whose nodes stand for `call`s, `success`es, `redo`s and `failure`s—i.e., the events which take place during execution. Classical LP visualisation tools, of which the Transparent Prolog Machine (TPM [9.13]) is paradigmatic, are based on this representation. In particular, the TPM uses an augmented AND-OR tree (AORTA), in which AND and OR branches are compressed and take up less vertical space, but the information conveyed by it is basically the same as in a usual AND-OR tree (see below a more complete description of the AORTA tree). This tree can be used both for displaying the search as explicitly coded by the programmer, and for representing the search implicitly performed by enumeration predicates.

It is true that in CLP programs the control part has usually less importance than in LP, since most of the time is spent in equation solving and enumeration. However, note that one of the main differences between C(L)P and, e.g., Operations Research, is the ability to set up equations in an algorithmic fashion, and to search for the right set of equations. Although this part may sometimes be short (and perhaps deterministic), it may also be

[1] By "fixed" we mean that these rules can be deterministically known at run-time, although maybe not known statically.

quite large and it is in any case relevant, for performance debugging, to be able to represent and understand its control flow.

Given the previous considerations, a first approach which can be used in order to visualise CLP executions is to represent the part corresponding to the execution of the program clauses (the programmed search) using a search-tree depiction. Note that the constraint-related operations of a CLP execution (enumeration/propagation) typically occur in "bursts" which can be associated to points of the search-tree. Thus, the search tree depiction can be seen as primary view or a skeleton onto which other views of the state of the constraint store during enumeration and propagation (and which we will address in Section 9.4) can be grafted or to which they can be related.

9.4 Representing the Enumeration Process

The enumeration process typically performed by finite domain solvers (involving, e.g., domain splitting, choosing paths for constraint propagation, and heuristics for enumeration) often affects performance critically. Observing the behaviour of this process in a given problem (or class of problems) can help to understand the source of performance flaws and reveal that a different set of constraints or a different enumeration strategy would improve the efficiency of the program.

The enumeration phase can be seen as a search for a mapping of values to variables which satisfy all the constraints set up so far. It takes the form of (or can be modelled as) a search which non-deterministically narrows the domains of the variables and, as a result of the propagation of these changes, updates the domains of other variables. Each of these steps results in either failure (in which case another branch of the search is chosen by setting the domain of the selected variable differently or by picking another variable to update) or in a new state with updated domains for the variables.[2] Thus, one approach in order to depict this process is to use the same representation proposed for the programmed search, i.e., to use a tree representation, in either time or event space. In this case nodes concerning the selection of variables and selection of domains should be clearly distinguished, as they represent radically different choices. Another alternative, which focuses more on data evolution than on control flow, is to simply visualise those steps as a succession of states for all the variables (as shown in Section 10.2.1 and Figures 10.3 and 10.4 of Chapter 10), or show an altogether *ad-hoc* representation of enumeration.

The display of the enumeration process can have different degrees of faithfulness to what actually happens internally in the solver. Showing the internal

[2] This enumeration can often be encoded as a Prolog-like search procedure which selects a variable, inspects its domain, and narrows it, with failure as a possible result. The inspection and setting of the domains of the variables are typically primitive operations of the underlying system.

behaviour of the solver is not always possible, since in some CLP systems the enumeration and propagation parts of the execution are performed at a level not accessible from user code. This complicates the program visualisation, since in order to gather data, either the system itself has to be instrumented to produce the data (as in the CHIP Tree Visualiser [9.1], Chapter 7), or sufficient knowledge about the solver operation must be available so that its operation can be mimicked externally in a meta-interpreter inside the visualiser, and inserted transparently between the user-perceived execution steps.

Other types of visualisation concerned with the internal work performed by the solver need low-level support from the constraint solver. They are very useful for system implementors who have access to the system internals, and for the programmer who wants to really fine-tune a program to achieve superior performance in a given platform, but its own nature prevents them from being portable across platforms. Therefore we chose to move towards generalisation at the expense of some losses, and base a more general, user-definable depiction, on simpler, portable primitives, while possible. These may not have access to all the internal characteristics of a programming system, but in turn can be used in a variety of environments.

9.5 Coupling Control Visualisation with Assertions

One of the techniques used frequently for program verification and correctness debugging is to use assertions which (partially) describe the specification and check the program against these assertions (see, e.g., [9.2, 9.10, 9.5], and Chapters 1, 2, 3, and 4 and their references). The program can sometimes be checked statically for compliance with the assertions, and when this cannot be ensured, run-time tests can be automatically incorporated into the program. Typically, a warning is issued if any of these run-time tests fail, flagging an error in the program, since it has reached a state not allowed by the specification. It appears useful to couple this kind of run-time testing with control visualisation. Nodes which correspond to run-time tests can be, for example, colour coded to reflect whether the associated check succeeded or failed; the latter case may not necessarily mean that the branch being executed has to fail as well. This allows the programmer to easily pinpoint the state of the execution that results in the violation of an assertion (and, thus, of the specification) and, by clicking on the nodes associated to the run-time checks, to explore for the reason of the error by following the source of instantiation of the variables. As mentioned before, the design of the tree is independent from the constraint domain, and so the user should be able to click on a node and bring up a window (perhaps under the control of a different application) which shows the variables / constraints active at the moment in which the node was clicked. This window allows the programmer

to peruse the state of the variables and detect which are the precise sources of the values of the variables involved in the faulty assertion.

This does not mean, of course, that assertion checking at compile time should be looked on as opposed to visualisation: rather, visualisation can be effectively used as user interface, with interesting characteristics of its own, to assertion-based debugging methods.

9.6 The APT Tool

In order to test the basic ideas of the previous sections, we extended the APT tool (*A Prolog Tracer* [9.17]) to serve as a CLP control visualiser. APT is essentially a TPM-based search-tree visualiser,[3] and inherits many characteristics from the TPM. However, APT also adds some interesting new features. APT is built around a meta-interpreter coded in Prolog which rewrites the source program and runs it, gathering information about the goals executed and the state of the store at run-time. This execution can be performed depth-first or breadth-first, and can be replayed at will, using the collected information. All APT windows are animated, and are updated as the (re)execution of the program proceeds.

```
a(X,Y):- b(X,Z), c(Z,Y).
a(X,Y):- X=Y.

b(1,2).

c(2,4).
```

Fig. 9.1. Sample code

The main visualisation of APT offers a tree-like depiction, in which nodes from calls to user code are represented by squares and are adorned optionally with the name of the predicate being called. Nodes corresponding to built-ins appear as circles. Figure 9.1 shows a small program, and Figure 9.2 the execution tree corresponding to this program with the query `a(2, 2)`. Goals in the body of the first clause of `a/2` (`b(X, Z)` and `c(Z,Y)`) are shown as nodes whose edges to their parent are crossed with a line — these are AND-branches corresponding to the goals inside the clause. The goals in the body of the second clause (only one, in this case) are linked to the parent node with a separate set of edges. The actual run-time arguments are not shown at this level, but nodes can be blown up for more detail, as we will see later.

[3] Another CLP visualiser that depicts the control part as a tree in the TPM spirit is the one developed by PrologIA [9.19], Chapter 6.

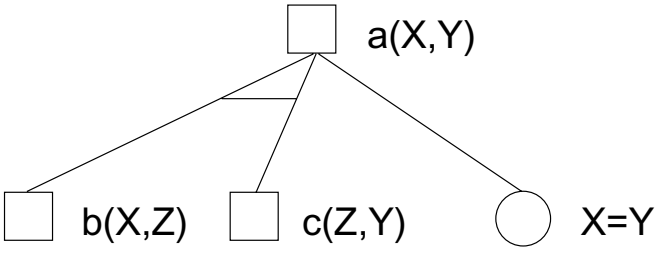


Fig. 9.2. AORTA execution tree for the program in Figure 9.1

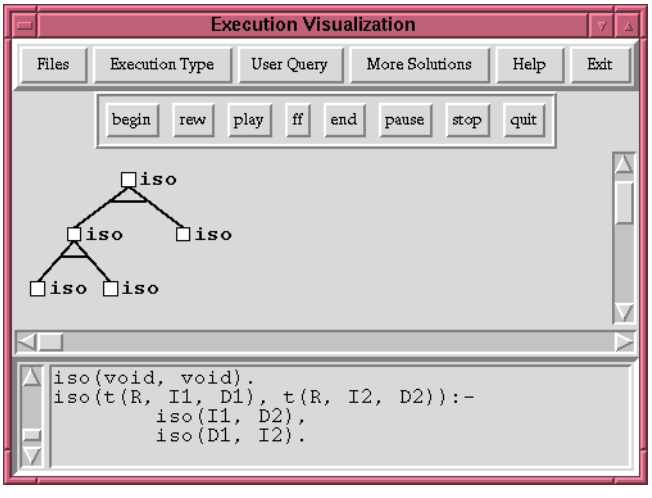


Fig. 9.3. A small execution tree, as shown by APT

Figure 9.3 shows a view of an actual APT window. In a real execution, the state of each node (*not yet called, called but not yet exited, exited, failed*) is shown by means of a colour code. Clicking on a node opens a different window in which the relevant part of the program source, i.e., the calling body atom and the matching clause head, is represented together with the (run-time) state of the variables in that node (Figure 9.4).

The presentation of these node views depends on the type of data (e.g., the constraint domain) used. This is one of the most useful general concepts underlying the design of APT: the graphical display of control is logically separated from that of data. This allows developing data visualisations independently from the control visualisation, and using them together. The data visualisation can then be taken care of by a variety of tools, depending on the data to be visualised. Following the proposal outlined in the previous section, this allows using APT as a control skeleton for visualising CLP execution. In

this case, the windows which are opened when clicking on the tree nodes offer views of the constraint store in the state represented by the selected node. These views vary depending on the constraint domain used, or even for the same domain, depending on the data visualisation paradigm used.

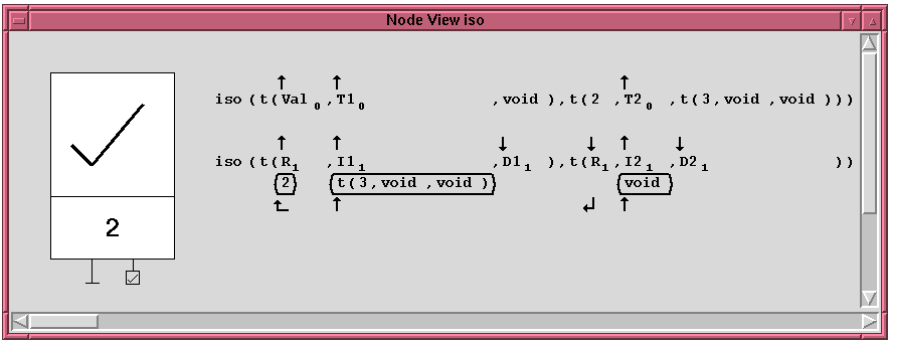


Fig. 9.4. Detailed view of a node (Herbrand domain)

The figure at the left of the program text (still in Figure 9.4) represents the state of the call: marked with a tick ($\surd$) for success (as in this case), crossed ($\times$) for failure, and signalled with a question mark if the call has not finished yet. The number below the symbol denotes the number of clauses in the predicate. For each of these clauses there is a small segment sticking out from the bottom of the box. Clauses tried and failed have a "bottom" ($\perp$) sign; the clause (if any) currently under execution, but not yet finished, has a small box with a question mark; and a clause finished with success is marked with a tick.

Visualisations for constraints and constrained data are discussed elsewhere (for example, in Chapters 10, 11, and 12, and their references). As an example of such a visualisation, Figure 9.4 shows a depiction used for the Herbrand domain (which is built into APT as the default node depiction, given that most CLP systems include the Herbrand domain). The node blow-up shows the run-time call on top and the matching head below it. The answer substitution (i.e., the result of head unification and/or body execution) is shown enclosed by rounded rectangles. The arrows represent the source and target of the substitution, i.e., the data flow. In the example shown, variable $\texttt{R}_1$ received a value `2` from a call, and this value is communicated through head unification to variable $\texttt{Val}_0$, which returns it to the caller. On the other hand, variable $\texttt{T1}_0$ in the call unifies with variable $\texttt{I1}_1$ in the head, and returns an output value `t(3, void, void)` which comes from some internal body call ($\texttt{I1}_1$ does not appear anywhere else in the head of the clause).

APT is able to show the origin of the instantiation of any variables at any moment in the execution. In order to do that, APT keeps track of the point in the tree in which the (current) substitution of a variable was generated. Clicking on a substitution causes a line in the main tree to be drawn from the current node to the node where the substitution was generated. This is a very powerful feature which helps in correctness debugging, as the source of a (presumably) wrong instantiation (causing, for example an unexpected failure or a wrong answer) can be easily located. The culprit node can in turn be blown up and inspected to find out the cause of the generation of those values.[4]

Some More Details on The APT Tool. The tool reads and executes programs, generating an internal trace with information about the search-tree, the variables in each call, and the run-time (Herbrand) constraints associated. The execution can then be replayed, either automatically or step-by-step, and the user can move forwards and backwards in time. More detailed information about each invocation can be requested. The tool has a built-in text editor, with a full range of editing commands, most of them compatible with Emacs. Files open from the visualiser are loaded into the editor.

Execution can be performed either in depth-first or breadth-first mode. In the case of depth-first search, the user can specify a maximum depth to search; when this depth has been reached, the user is warned and prompted to decide whether to stop executing, or to search with a new, deeper maximum level. The search mode and search depth are controlled by the meta-interpreter built in APT, so that no special characteristic is required from the underlying CLP system.

The queries to the program are entered in the window APT was launched from. Once a query has been finished, the user can ask for another solution to the same query. As in a top-level, this is performed by forcing backtracking after a simulated failure. If the tree to be visualised is too large to fit in the window (which is often the case), slide bars make it possible to navigate through the execution.

APT uses Tcl/Tk [9.18] to provide the graphical interface. The original implementation of APT was developed under SICStus Prolog. It has also been ported to the clp(fd)/Calypso system developed at INRIA [9.9].

9.7 Event-Based and Time-Based Depiction of Control

In our experience, tree-based representations such as those of the TPM, APT, and similar tools are certainly quite useful in education and for correctness

[4] APT uses a "rich" meta-interpreter, in the sense that it keeps track of a large amount of information. In retrospect, the "rich meta-interpreter" approach has advantages and disadvantages. On one hand it allows determining very interesting information such, e.g., the origin of a given binding mentioned previously. On the other hand, it cannot cope with large executions.

and performance debugging. In some cases, the shape of the search-tree can help in tracking down sources of unexpected low performance, showing, for example, which computation patterns have been executed more often, or which parts dominate the execution. However, the lack of a representation of time (or, in general, of resource consumption) greatly hinders the use of simple search-trees in performance debugging. AND-OR trees, as those used in APT, do not depict usually time (or in general, resource) consumption; they need to be adorned with more information.

One approach in order to remedy this is to incorporate resource-related information into the depiction itself, for example by making the distance between a node and its children reflect the elapsed time (or amount of resource consumed). Such a representation in *time space* provides insight into the cost of different parts of the execution: in a CLP language not all user-perceived steps have the same cost, and therefore they should be represented with a different associated height. For example, constraint addition, removal, unification, backtracking, etc. can have different associated penalties for different programs, and, even for the same program, the very same operation can cause a different overhead at different points in the execution.

Time-oriented views have been used in several other (C)LP visualisation tools, such as VisAndOr [9.7] (included with recent distributions of SICStus

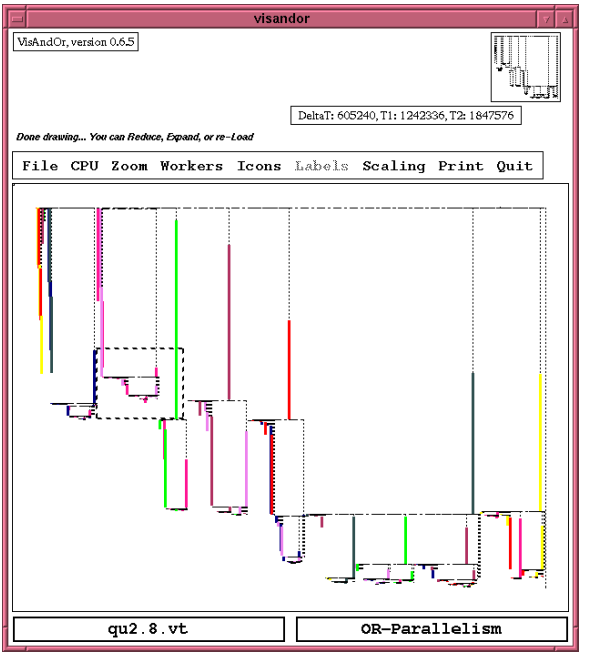


Fig. 9.5. VisAndOr showing an execution in time space

Fig. 9.6. VisAndOr showing an execution in event space

[9.21]) and VISTA [9.22]. VisAndOr is a graphical tool aimed at displaying and understanding the performance of parallel execution of logic programs, while VISTA focuses on concurrent logic programs. In VisAndOr, time runs from top to bottom, and parallel tasks are drawn as vertical lines (see Figure 9.5). These lines use different colours and thickness to represent which processor executes each task and the state of the task: running, waiting to be executed, or finished. In this case, the length of the vertical lines reflects accurately a measure of the time spent.

A VisAndOr view can be described as an skeletal depiction of a logic program execution in which only nodes with relevance to the parallelism (forks & joins) were chosen to be displayed. And, as an interesting feature, the tree is adorned with tags (colours and line thickness) which add information without cluttering the display. VisAndOr allows also switching to an *event space*, in which every event in the execution (say, the creation, the start, and the end of a task, among others) takes the same amount of space—see Figure 9.6, where the same execution as in Figure 9.5 is depicted. Note that in this view the structure of the execution is easier to see, but the notion of time is lost—or, better, traded off for an alternate view. This event-oriented visualisation is the one usually portrayed in the tree-like representation for the execution of logic programs: events are associated to the `calls` made in

the program, and space is evenly divided among those events. Thus, event- and time-based visualisation are not exclusive, but rather complementary to each other, and it is worth having both in a visualisation tool aimed at program debugging.

9.8 Abstracting Control

The search-tree discussed throughout this chapter gives a good representation of the space being traversed. It also offers some degree of abstraction with respect to a classical search-tree by reusing the tree nodes during backtracking. But it has the drawback of being too explicit, taking up too much space, and showing too much detail to be useful in medium-sized computations, which can easily generate thousands of nodes. A means of abstracting this view is desirable.

An obvious way to cope with a very large number of objects (nodes and links) in the limited space provided by a screen is using a virtual canvas larger than the physical screen (as done by APT). However, this makes it difficult to perceive the "big picture". An alternative is simply squeezing the picture to fit into the available space; this can be made uniformly, or with a selection which changes the compression ratio in different parts of the image (this, in fact, is related to whether a time- or event-oriented view is used).

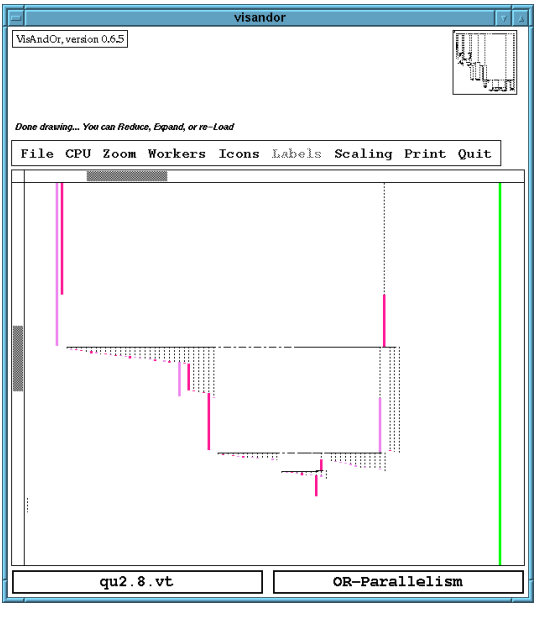


Fig. 9.7. Zoomed view in VisAndOr

The former has the drawback that we lose the capability to see the details of the execution when necessary. The latter seems more promising, since there might be parts of the tree which the user is not really interested in watching in detail (for example, because they belong to parts of the program which have already been tested).

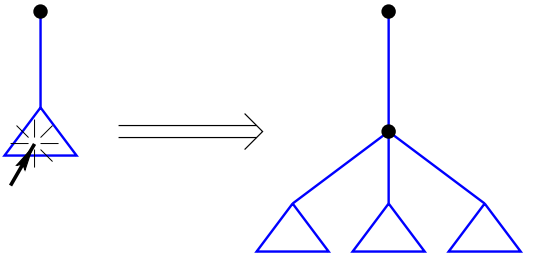


Fig. 9.8. Exposing hidden parts of a tree

An example of a tool which compresses automatically parts of the search-tree is the VISTA tool for the visualisation of concurrent logic programs [9.22]. This compression is performed automatically at the points of greater density of objects—near the leaves. But this disallows blowing up those parts if a greater detail is needed. An alternative possibility is to allow the user to slide virtual magnifying lenses, which provide with a sort of fish-eye transformation and give both a global view (because the whole tree is shrunk to fit in a window) and a detailed view (because selected parts of the tree are zoomed out to greater detail). Providing at the same time a compressed view of the whole search-tree, in which the area being zoomed is clearly depicted, can also help to locate the place we are looking at; this option was already present in VisAndOr, where the canvas could be zoomed out, and the window on it was represented as a dotted square in a reduced view of the whole execution (Figure 9.7, corresponding to the dotted square in the top right corner of Figure 9.5).

Another possibility to avoid cluttering up the display is to allow the user to hide parts of the tree (see Figure 9.8 and [9.20]). This actually allows for the selective exploration of the tree (i.e., in the cases where a call is being made to a predicate known to be correct, or whose performance has already been tested). Whereas this avoids having too many objects at a time, feedback on the relative sizes of the subtrees is lost. It can be recovered, though, by tagging the collapsed subtrees with a mark which measures the relative importance of the subtrees. This "importance" can range from execution time to the number of nodes, number of calls, number of added constraints, number of fix-point steps in the solver, etc.; different measures would lead to different abstraction points of view. Possible tagging schemes are raw numbers

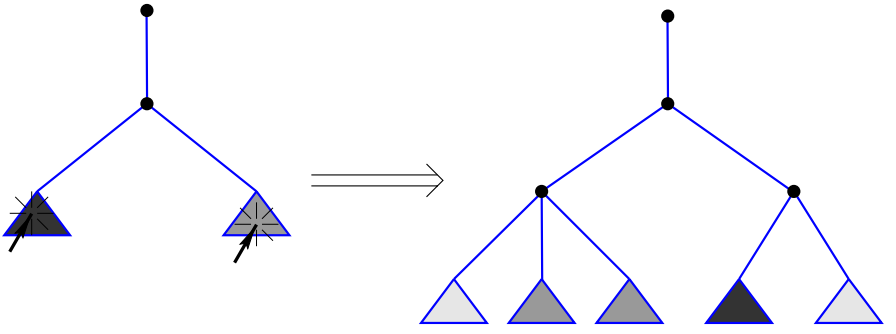


Fig. 9.9. Abstracting parts of a tree

attached to the collapsed subtrees (indicating the concrete value measured under the subtree) or different shades of gray (which should be automatically re-scaled as subtrees are collapsed/expanded; see Figure 9.9). Representations of tree abstractions are currently being incorporated into APT.

9.9 Conclusions

We have presented some design considerations regarding the depiction of search-trees resulting from the execution of constraint logic programs. We have argued that these depictions are applicable to the programmed search part and to the enumeration parts of such executions. We have also presented a concrete tool, APT, based on these ideas. Two interesting characteristics of this tool are, first, the decoupling of the representation of control from the constraint domain used in the program (and from the representation of the store), and, second, the recording of the point in which every variable is created and assigned a value. The former allows visualisers for variables and constraints in different domains (such as those presented in Chapter 10) to be plugged and used when necessary. The latter allows tracking the source of unexpected values and failures.

APT has served mainly as an experimentation prototype for, on one hand, studying the viability of some of the depictions proposed, and, on the other, as a skeleton on which the constraint-level views presented in the following chapters can be attached. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed by Cosytec for the CHIP system, and which are described in other chapters.

Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments. Also thanks to Abder Aggoun and Helmut Simonis and other DiSCiPl project members for many discussions on constraint visualisation.

The prototype implementation of APT was done by Ángel López [9.17].

The material in this chapter comes mainly from DiSCiPl project reports, parts of which were already presented at the AGP'98 Joint Conference on Declarative Programming [9.8].

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

9.1 A. Aggoun and H. Simonis. Search Tree Visualization. Technical Report D.WP1.1.M1.1-2, COSYTEC, June 1997. In the ESPRIT LTR Project 22352 DiSCiPl.

9.2 K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

9.3 R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

9.4 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.

9.5 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, U. of Linköping Press, May 1997.

9.6 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

9.7 M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201, The MIT Press, June 1993.

9.8 M. Carro and M. Hermenegildo. Some Design Issues in the Visualization of Constraint Program Execution. In *AGP'98 Joint Conference on Declarative Programming*, pages 71–86, July 1998.

9.9 D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, The MIT Press, 1993.

9.10 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522, The MIT Press, 1989.

9.11 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

9.12 M. Ducassé. A General Query Mechanism Based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Programming Language Implementation and Logic Programming, PLILP'92*, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, 1992.
9.13 M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.
9.14 M. Fabris. CP Debugging Needs. Technical report, ICON s.r.l., April 1997. ESPRIT LTR Project 22352 DiSCiPl deliverable D.WP1.1.M1.1.
9.15 J.M. Fernández. Declarative debugging for babel. Master's thesis, School of Computer Science, Technical University of Madrid, October 1994.
9.16 K. Kahn. Drawing on Napkins, Video-game Animation, and Other ways to program Computers. *Communications of the ACM*, 39(8):49–59, August 1996.
9.17 A. López Luengo. Apt: Implementing a graphical visualizer of the execution of logic programs. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, October 1997.
9.18 J. K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.
9.19 PrologIA. Visual tools for debugging of Prolog IV programs. Technical Report D.WP3.5.M2.2, ESPRIT LTR Project 22352 DiSCiPl, October 1998.
9.20 C. Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *ICLP'97*, TheMIT Press, July 1997.
9.21 Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.
9.22 E. Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.

10. Tools for Constraint Visualisation: The VIFID/TRIFID Tool

Manuel Carro and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, S-28660-Madrid, Spain
email: {mcarro,herme}@fi.upm.es

Visualisation of program executions has been used in applications which include education and debugging. However, traditional visualisation techniques often fall short of expectations or are altogether inadequate for new programming paradigms, such as Constraint Logic Programming (CLP), whose declarative and operational semantics differ in some crucial ways from those of other paradigms. In particular, traditional ideas regarding the behaviour of data often cannot be lifted in a straightforward way to (C)LP from other families of programming languages. In this chapter we discuss techniques for visualising data evolution in CLP. We briefly review some previously proposed visualisation paradigms, and also propose a number of (to our knowledge) novel ones. The graphical representations have been chosen based on the perceived needs of a programmer trying to analyse the behaviour and characteristics of an execution. In particular, we concentrate on the representation of the run-time values of the variables, and the constraints among them. Given our interest in visualising large executions, we also pay attention to abstraction techniques, i.e., techniques which are intended to help in reducing the complexity of the visual information.

10.1 Introduction

As mentioned in the previous chapter, program visualisation has focused classically on the representation of program flow (using flowcharts and block diagrams, for example) or on the data manipulated by the program and its evolution as the program is executed. In that chapter we discussed issues related to the visualisation of the "programmed control" or "programmed search" part of the execution of constraint logic programs [10.17, 10.19]. In this chapter we focus on methods for displaying the contents of variables, the constraints among such variables, the evolution of such contents and constraints, and abstractions of the proposed depictions. For simplicity, and because of their relevance in practice, in what follows we will discuss mainly the representation of finite domain (FD) constraints and variables, although we will also mention other constraint domains and consider how the visualisations designed herein can be applied to them.

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 253–272, 2000.

10.2 Displaying Variables

In imperative and functional programming there is a clear notion of the values that variables are bound to (although it is indeed more complex in the case of higher-order functional variables). The concept of variable binding in LP is somewhat more complex, due to the variable sharing which may occur among Herbrand terms. The problem is even more complex in the case of CLP, where such sharing is generalised to the form of equations relating variables. As a result, the value of C(L)P variables often is actually a complex object representing the fact that each variable can take a (potentially infinite) set of values, and that there are constraints attached to such variables which relate them and which restrict the values they can take simultaneously.

Textual representations of the variables in the store are usually not very informative and difficult to interpret and understand.[1] A graphical depiction of the values of the variables can offer a view of computation states that is easier to grasp. Also, if we wish to follow the history of the program (which is another way of understanding the program behaviour, but focusing on the data evolution), it is desirable that the graphical representation be either animated (i.e., time in the program is depicted in the visualisation also as time) or laid out spatially as a succession of pictures. The latter allows comparing different behaviours easily, trading time for space.

Since different constraint domains have different properties and characteristics, different representations for variables may be needed for them. In what follows we will sketch some ideas focusing on the representation of variables in Finite Domains, but we will also refer briefly to the depiction of other commonly used domains.

10.2.1 Depicting Finite Domain Variables

As mentioned before, Finite Domains (FD) are one of the most popular constraint domains. FD variables take values over finite sets of integers which are the domains of such variables. The operations allowed among FD variables are pointwise extensions of common integer arithmetic operations, and the allowed constraints are the pointwise variants of arithmetic constraints. At any state in the execution, each FD variable has an active domain (the set of allowed values for it) which is usually accessible by using primitives of the language. For efficiency reasons, in practical systems this domain is usually an upper approximation of the actual set of values that the variable can theoretically take. We will return to this characteristic later, and we will see how taking it into account is necessary in order to obtain correct depictions of values of variables.

A possible graphical representation for the state of FD variables is to assign a dot (or, depending on the visualisation desired, a square) to every

[1] Also note that some solvers maintain, for efficiency or accuracy reasons, only an approximation of the values the variables can take.

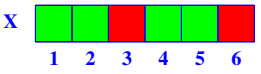


Fig. 10.1. Depiction of a finite domain variable

Fig. 10.2. Shades representing age of discarded values

possible value the variable can take; therefore the whole domain is a line (respectively, a rectangle). Values belonging to the current domain at every moment are highlighted. An example of the representation of a variable X with current domain $\{1, 2, 4, 5\}$ from an initial domain $\{1 \ldots 6\}$ is shown in Figure 10.1. More possibilities include using different colours / shades / textures to represent more information about the values, as in Figure 10.2 (this is done also, for example, in the GRACE visualiser [10.20]).

Looking at the static values of variables at only one point in the execution (for example, the final state) obviously does not provide much information on how the execution has actually progressed. However, the idea is that such a representation can be associated with each of the nodes of the control tree, as suggested in Chapter 9, i.e., the window that is opened upon clicking on a node in the search-tree contains a graphical visualisation for each of the variables that are relevant to that node. The variables involved can be represented in principle simply side to side as in Figure 10.6 (we will discuss how to represent the relations between variables, i.e., constraints, in Section 10.3).

Note that each node of the search-tree often represents several internal steps in the solver (e.g., propagation is not seen by the user, or reflected in user code). The visualisation associated to a node can thus represent either the final state of the solver operations that correspond to that search-tree node, or the history of the involved variables through all the internal solver (or enumeration) steps corresponding to that node.

Also, in some cases, it may be useful to follow the evolution of a set of program variables throughout the program execution, independently of what node in the search-tree they correspond to (this is done, for example, in some of the visualisation tools for CHIP [10.1]). This also requires a depiction of the values of a set of variables over time, and the same solutions used for the previous case can be used.

Thus, it is interesting to have some way of depicting the evolution in time of the values of several variables. A number of approaches can be used to achieve this:

– An animated display which follows the update of the (selected) variables step by step as it happens; in this case, time is represented as time. This makes the immediate comparison of two different stages of the execution difficult, since it requires repeatedly going back and forth in time. However, the advantage is that the representation is compact and can be useful for understanding how the domains of the variables are narrowed. We will return to this approach later.

– Different shadings (or hues of colour) can be used in the boxes corresponding to the values, representing in some way how long ago that value has been removed from the domain of the variable (see Figure 10.2, where darker squares represent values removed longer ago). Unfortunately, comparing shades accurately is not easy for the human eye, although it may give a rough and very compact indication of the changes in the history of the variable. An easier to interpret representation would probably involve adjusting the shades so that the human brain interprets them correctly when squares of different shades surround it.
– A third solution is to simply stack the different state representations, as in Figure 10.3. This depiction can be easily shrunk/scrolled if needed to accommodate the whole variable history in a given space. It can represent time accurately (for example, by reflecting it in the height between changes) or ignore it, working then in *events space* (see Chapter 9), by simply stacking a new line of a constant height every time a variable domain changes, or every time an enumeration step is performed. This representation allows the user to perform an easier comparison between states and has the additional advantage of allowing more time-related information to be added to the display.

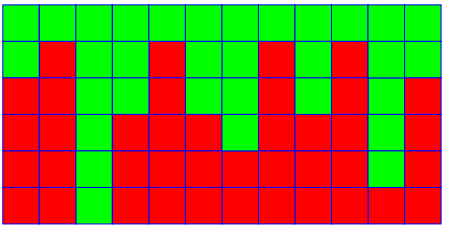


Fig. 10.3. History of a single variable (same as in Figure 10.2)

The last approach is one of the visualisations available in the VIFID visualiser, implemented at UPM, and which, given a set of variables in a FD program, generates windows which display states (or sets of states) for those variables. VIFID can be used as a visualiser of the state in nodes of the search-tree, or stand alone, as a user library, in which case the display is triggered by spy-points introduced by the user in the program. Figure 10.15 shows an example of such an annotated program, where the lines calling `log_state/1` implement the spy-points, and Figure 10.4 shows a screen dump of a window generated by VIFID presenting the evolution of selected program variables in a program to solve the queens problem for a board of size 10. Each column in the display corresponds to one program variable, and are labelled with the name of the variables on top. In this case the possible values are the row numbers in which a queen can be placed. Lighter squares represent

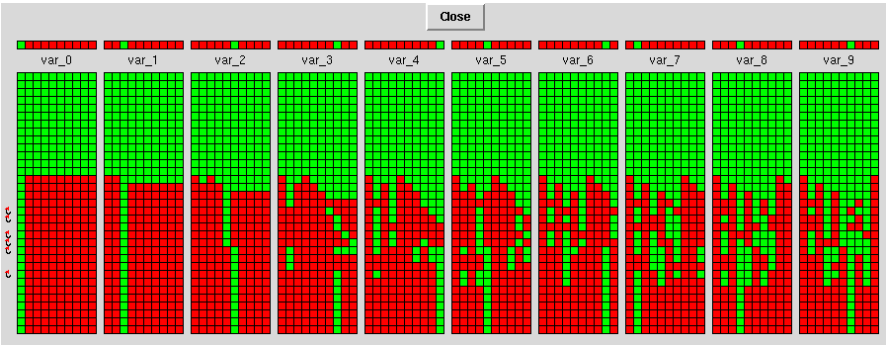


Fig. 10.4. Evolution of FD variables for a 10 queens problem

values still in the domain, and darker squares represent discarded values. Each row in the display corresponds to a spy-point in the source program, which caused VIFID to consult the store and update the visualisation. Points where backtracking took place are marked with small curved arrows pointing upwards. It is quite easy to see that very little backtracking was necessary, and that variables are highly constrained, so that enumeration (proceeding left to right) quite quickly discarded initial values. VIFID supports several other visualisations, some of which will be presented later in the chapter.

Some of the problems which appear in a display of this type are the possibly large number of variables to be represented and the size of the domain of each variable. Note that the first problem is under control to some extent in the approach proposed: if the visualisation is simply triggered from a selected node in the search-tree, the display can be forced to present only the relevant variables (e.g., the ones in the clause corresponding to that node). In the case of triggering the visualisation through spy-points in the user program, the number of variables is under user control, since they are selected explicitly when introducing the spy-points. The size of the domains of variables is more difficult to control (we return to this issue in Section 10.4.1). However, note that, without loss of generality, programs using FD variables can be assumed to initialise the variables to an integer range which includes all the possible values allowable in the state corresponding to the beginning of the program.[2] However, being able to deduce a small initial domain for a variable allows starting from a more compact initial representation for that variable. This in turn will allow a more compact depiction of the narrowing of the range of the variable, and of how values are discarded as the execution proceeds. Other abstraction means for coping with large executions are discussed in Section 10.4.1.

[2] In the default case, variables can be assumed to be initialised to the whole domain.

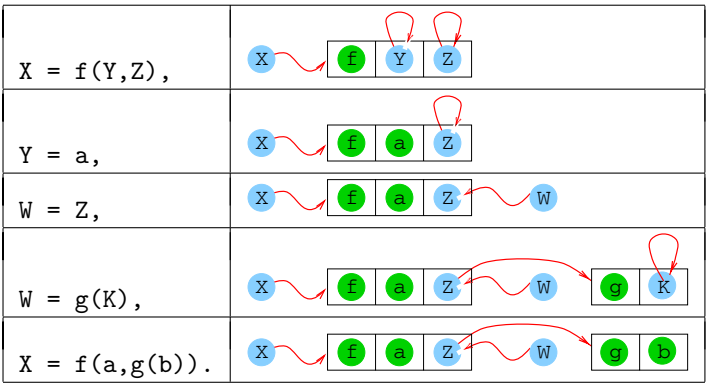


Fig. 10.5. Alternative depiction of the creation of a Herbrand term

10.2.2 Depicting Herbrand Terms

Herbrand terms can always be written textually, or with a slightly enhanced textual representation. An example is the depiction of nodes in the APT tool, discussed in Chapter 9. They can also be represented graphically, typically as trees. A term whose main functor is of arity n is then represented as a tree in which the root is the name of this functor, and the n subtrees are the trees corresponding to its arguments. This representation is well suited for ground terms. However, free variables, which may be shared by different terms, need to be represented in a special way. A possibility is to represent this sharing as just another edge (thus transforming the tree into an acyclic graph), and even, taking an approach closer to usual implementation designs, having a free variable to point to itself. This corresponds to a view of Herbrand terms as complex data structures with single assignment pointers. Figure 10.5 shows a representation using this view of the step by step creation of a complex Herbrand term by a succession of Herbrand constraints. Rational trees (as those supported by Prolog II, III, IV) can also be represented in a similar way—but in this case the graph can contain cycles, although it cannot be a general graph.

10.2.3 Depicting Intervals or Reals

In a broad sense, intervals resemble finite domains: the constraints and operations allowed in them are analogous (pointwise extensions of arithmetic operations), but the (theoretical) set of values allowed is continuous, which means that an infinite set of values are possible, even within a finite range. Despite these differences, visual representations similar to those proposed for finite domains can be easily used for interval variables, using a continuous line instead of a discrete set of squares. An important difference between intervals and finite domains is that intervals usually allow non-linear arithmetic

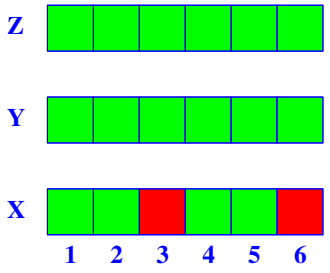


Fig. 10.6. Several variables side to side

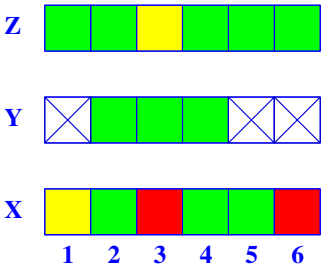


Fig. 10.7. Changing a domain

operations for which a solution procedure is not known, which forces the solvers to be incomplete. Thus, the visualisation of the actual domain[3] will in general be an upper approximation of the actual (mathematical) domain. As a result, an exact display of the intervals is not possible in practice. But the approach for showing the evolution in time (Figure 10.4) and for representing the constraints (Section 10.3) is still valid, although in the former case some means for dealing with phenomena inherent to solving in real-valued intervals (e.g., slow convergence of algorithms) should be taken.

10.3 Representing Constraints

In the previous section we have dealt with representations of the values of individual variables. It is obviously also interesting to represent the relationships among several variables as imposed by the constraints affecting them. This can sometimes be done textually by simply dumping the constraints and the variables involved in the source code representation. Unfortunately, this is often not straightforward (or even possible in some constraint domains), can be computationally expensive, and provides too much level of detail for an intuitive understanding.

Constraint visualisation can be used alternatively to provide information about which variables are interrelated by constraints, and how these interrelations make those variables affect each other. Obviously, classical geometric representations are a possible solution: for example, linear constraints can be represented geometrically with dots, lines, planes, etc., and nonlinear ones by curves, surfaces, volumes, etc. Standard mathematical packages can be used for this purpose. However, these representations are not without problems: working out the representation can be computationally expensive, and, due to the large number of variables involved the representations can easily be n-dimensional, with $n \gg 3$.

[3] Not only the representation, but also the internal representation, from which the graphical depiction is drawn.

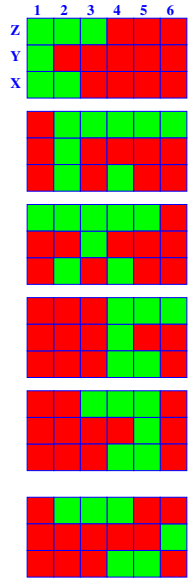


Fig. 10.8. Enumerating `Y`, representing solver domains `X` and `Z`

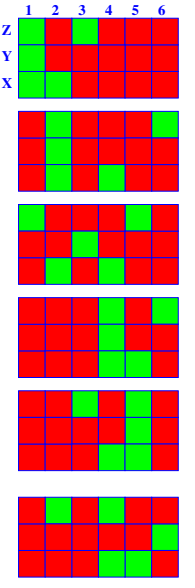


Fig. 10.9. Enumerating `Y`, representing also the enumerated domains for `X` and `Z`

A general solution which takes advantage of the representation of the actual values of a variable (and which is independent of how this representation is actually performed) is to use projections to present the data piecemeal and to allow the user to update the values of the variables that have been projected out, while observing how the variables being represented are affected by such changes. This can often provide the user with an intuition of the relationships linking the variables (and detect, for example, the presence of erroneous constraints). The update of these variables can be performed interactively by using the graphical interface (e.g., via a sliding bar), or adding manually a constraint, using the source CLP language.

We will use the constraint **C1**, below, in the examples which follow:

$$\mathbf{C1} : X \in \{1..6\} \wedge X \neq 6 \wedge X \neq 3 \wedge Z \in \{1..6\} \wedge Z = 2X - Y \wedge Y \in \{1..6\} \tag{10.1}$$

Figure 10.6 shows the actual domains of FD variables `X`, `Y`, and `Z` subject to the constraint **C1**. As before, lighter boxes represent points inside the domain of the variable, and darker boxes stand for values not compatible with the constraint(s). This representation allows the programmer to explore how changes in the domain of one variable affect the others: an update of the domain of a variable should indicate changes in the domains of other variables related to it. For example, we may discard the values 1, 5, and 6

from the domain of `Y`, which boils down to representing the constraint **C2**:

$$\mathbf{C2} : \mathbf{C1} \wedge Y \neq 1 \wedge Y < 5 \tag{10.2}$$

Figure 10.7 represents the new domains of the variables. Values directly disallowed by the user are shown as crossed boxes; values discarded by the effect of this constraint are shown in a lighter shade. In this example the domains of both `X` and `Z` are affected by this change, and so they depend on `Y`. This type of visualisation (with the two enumeration variants which we will comment on in the following paragraphs) is also available in the VIFID tool.

Within this same visualisation, a more detailed inspection can be done by leaving just one element in the domain of `Y`, and watching how the domains of `X` and `Z` are updated. In Figures 10.8 and 10.9 `Y` is given a definite value from `1` (in the topmost rectangle) to `6` (in the bottommost one). This allows the programmer to check that simple constraints hold among variables, or that more complex properties (e.g., that a variable is made definite by the definiteness of another one) are met.

The difference between the two figures lies in how values are determined to belong to the domain of the variable. In Figure 10.8, the values for `X` and `Z` are those kept internally by the solver, and are thus probably a safe approximation. In Figure 10.9, the corresponding values were obtained by enumerating `X` and `Z`, and the domains are smaller. Both figures were obtained using the same constraint solver, and comparing them gives an idea of how accurately the solver keeps the values of the variables. For several reasons (limitation of internal representation, speed of addition/removal of constraints, etc), quite often solvers do not keep the domains of the variables as accurately as it is possible. Over-constraining a problem may then help in causing an earlier failure. On the other hand, over-constraining increments the number of constraints to be processed and the time associated to this processing. Comparing the solver-based against the enumeration-based representation of variables helps in deciding whether there is room for improvement by adding redundant constraints.

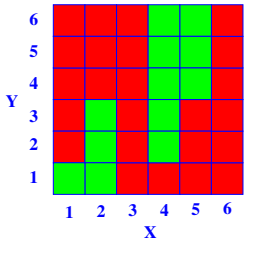


Fig. 10.10. `X` against `Y`

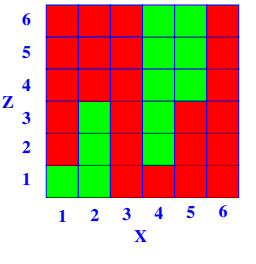


Fig. 10.11. `X` against `Z`

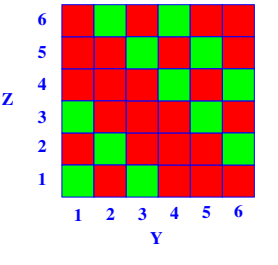


Fig. 10.12. `Y` against `Z`

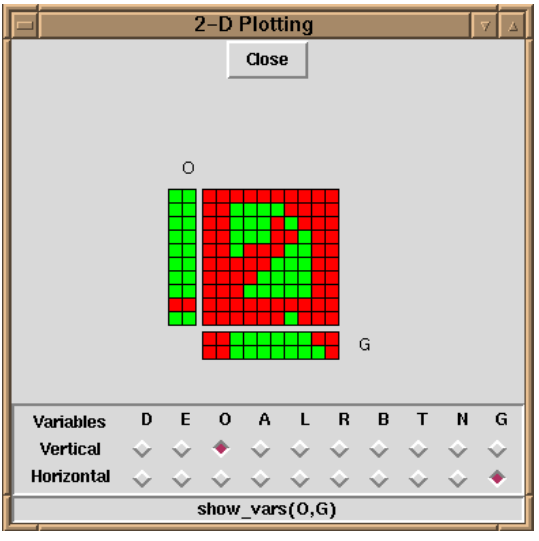


Fig. 10.13. Relating variables in VIFID

A static version of this view can be obtained by plotting values of pairs of variables in a 2-D grid, which is equivalent to choosing values for one of them and looking at the allowed values for the other. This is schematically shown in Figures 10.10, 10.11, and 10.12, where the variables are subject to the constraint **C2**. In each of these three figures we have represented a different pair of variables. From these representations we can deduce that the values `X = 3` and `X = 6` are not feasible, regardless the values of `Y` and `Z`. It turns out also that the plots of `X` against `Y` and `X` against `Z` (Figures 10.10 and 10.11) are identical. From this, one might guess that perhaps `Y` and `Z` have necessarily the same value, i.e., that the constraint `Z = Y` is enforced by the store. This possibility is discarded by Figure 10.12, in which we see that there are values of `Z` and `Y` which are not the same, and which in fact correspond to different values of `X`. Furthermore, the slope of the highlighted squares on the grid suggests that there is an inverse relationship between `Z` and `Y`: incrementing one of them would presumably decrement the other—and this is actually the case, from constraint **C1**. A VIFID window showing a 2-D plot appears in Figure 10.13; the check buttons at the bottom allow the user to select the variables to depict.

Note that, in principle, more than two variables could be depicted at the same time: for example, for three variables a 3-D depiction of a "Lego object" made out of cubes could be used. Navigating through such a representation (for example, by means of rotations and *virtual tours*), does not pose big implementation problems on the graphical side, but it may not necessarily give information as intuitively as the 2-D representation. The usefulness of

such a 3-D (or n-D) representation is still a topic of further research—but 3-D portraits of other representations are possible; see Section 10.4.2. On the other hand, we have found very useful the possibility of changing the value of one (or several) variables not plotted in the 2-D grid, and examine how this affects the values of the current domains of the plotted variables.

10.4 Abstraction

While representations which reflect all the data available in an execution can be acceptable (and even didactic) for "toy programs," it is often the case that they result in too much data being displayed for larger programs. Even if an easy-to-understand depiction is provided, the amount of data can overwhelm the user with an unwanted level of detail, and with the burden of having to navigate through it. This can be alleviated by *abstracting* the information presented. Here, "abstracting" refers to a process which allows a user to focus on interesting properties of the data available. Different abstraction levels and/or techniques can in principle be applied to any of the aforementioned graphical depictions, depending on which property is to be highlighted.

Note that the depictions presented so far already incorporate some abstractions: when using VIFID, the user selects the interesting variables and program points via the spy-points and the window controls. If it is interfaced with a tree representation tool (as, for example, the one presented in Chapter 9), the variables to visualise come naturally from those in the selected nodes. In what follows we will present several other ideas for performing abstraction, applied to the graphical representations we have discussed so far. Also, some new representations, which are not directly based on a refinement of others already presented, will be discussed.

10.4.1 Abstracting Values

While the problem of the presence of a large number of variables can be solved, at least in part, by the selection of interesting variables (a task that is difficult in itself), another problem remains: in the case of variables with a large number of possible values, representations such as those proposed in Section 10.2.1 can convey information too detailed to be really useful. At the limit, the screen resolution may be insufficient to assign a pixel to every single value in the domain, thus imposing an aliasing effect which would prevent reflecting faithfully the structure of the domain of the variable. This is easily solved by using standard techniques such as a canvas that is larger than the window, and scrollbars, providing means for zooming in and out, etc. A "fish-eye" technique can also be of help, giving the user the possibility of zooming precisely those parts which are more interesting, while at the same time trying to keep as much information as possible condensed in a

limited space. However, these methods are more "physical" approaches than true conceptual abstractions of the information, which are richer and more flexible.

An alternative is to perform a more semantic "compaction" of parts of the domain. As an example of such a compaction, which can be performed automatically, consider associating consecutive values in the domain of a variable to an interval (the smallest one enclosing those values) and representing this interval by a reduced number of points. A coarser-level solution, complementary to the graphical representation, is to present the domain of a variable simply as a number, denoting how many values remain in its current domain, thus providing an indication of its "degree of freedom". A similar approach can be applied to interval variables, using the difference between the maximum and minimum values in their domains, or the total length of the intervals in their domains.

Another alternative for abstraction is to use an application-oriented filtering of the variable domains. For example, if some parts of the program are trusted to be correct, their effects in the constraint store can be masked out by removing the values already discarded from the representation of the variables, thus leaving less values to be depicted. E.g., if a variable is known to take only odd values, the even values are simply not shown in the representation. This filtering can be specified using the source language—in fact, the constraint which is to be abstracted should be the filter of the domain of the displayed variables.

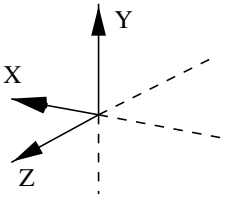
Note that this transformation of the domain cannot be completely automated easily: the debugger may not have any way of knowing which parts of the program are trusted and which are not, or which abstraction should be applied to a given problem. Thus, the user should indicate, with annotations in the program [10.9, 10.4] or interactively, which constraints should be used to abstract the variable values. Given this information, the actual reduction of the representation can be accomplished automatically. Warnings could be issued by the debugger if the values discarded by the program do not correspond to those that the user (or the annotations in the program) want to remove: if this happens, a sort of "out of domain" condition can be raised. This condition does not mean necessarily that there is an error: the user may choose not to show uninteresting values which were not (yet) removed by the program.

10.4.2 Domain Compaction and New Dimensions

Besides the problems in applications with large domains, the static representations of the history of the execution (Figure 10.4) can also fall short in showing intuitively how variables converge towards their final values, again because of the excess of points in the domains, or because an execution shows a "chaotic" profile. The previously proposed solution of using the *domain size*

as an abstraction can be applied here too. However, using raw numbers directly in order to represent this abstraction to the user is not very useful because it is not easy for humans to visualise arrays of numbers. A possible solution is to resort to shades of gray, but this may once again not work too well in practice: deducing a structure from a picture composed of different levels of brightness is not straightforward, and the situation may get even worse if colours are added.

A better option is to use the number of active values in the domain as coordinates in an additional dimension, thus leading to a 3-D visualisation. A possible meaning of each of the dimensions in such a representation appears in Figure 10.14. As in Figure 10.4, two axes correspond to time and selected variables: time runs along the **Z** direction, and every row along this dimension corresponds to a snapshot of the set of FD variables which have been selected for visualisation. In each of these rows, the size of the domain of the variable (according to the internal representation of the solver) is depicted as the dimension **Y**.



X The CLP(FD) variables.

Y An abstraction of the variable: the size of its domain.

Z Time.

Fig. 10.14. Meaning of the dimensions in the 3-D representation.

Figure 10.15 shows a CLP(FD) program for the `DONALD + GERALD = ROBERT` puzzle; we will inspect the behaviour of this program using two different orderings, defined by the `order/3` predicate. A different series of choices concerning variables and values (and, thus, a different search-tree) will be generated by using each of these options. The program (including the labelling routines) was annotated with calls to predicates which act as spy-points, and log the sizes of the domains of each variable (and, maybe, other information pertaining to the state of the program) at the time of each call. This information is unaffected by backtracking, and thus it can also keep information about the choices made during the execution. The `Handle` used to log the data may point to an internal database, an external file, or even a socket-based connection for on-line visualisation or even remote debugging.

Figure 10.16 is an execution of the program in Figure 10.15, using the first ordering of the variables. The variables closer to the origin (the ones which were labelled first) are assigned values quite soon in the execution and they remain fixed. But there are backtracking points scattered along the execution, which appear as blocks of variables protruding out of the picture. There is also a variable (which can be viewed as a white strip in the middle

```
:- use_module(library(clpfd)).
:- use_module(library(visclpfd)).

dgr(WhichOrder, ListOfVars):-
    ListOfVars = [D,O,N,A,L,G,E,R,B,T]
    order(WhichOrder, ListOfVars, OrderedVars),          %% Added
    open_log(dgr, ListOfVars, Handle),                   %% Added
    domain(OrderedVars, 0, 9),
    log_state(Handle),                                   %% Added
    D #> 0,
    log_state(Handle),                                   %% Added
    G #> 0,
    log_state(Handle),                                   %% Added
    all_different(OrderedVars),
    log_state(Handle),                                   %% Added
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
    100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
    100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
    log_state(Handle),                                   %% Added
    user_labeling(OrderedVars, Handle),
    close_log(Handle).

order(1, [D,O,N,A,L,G,E,R,B,T], [D,G,R,O,E,N,B,A,L,T]).
order(2, [D,O,N,A,L,G,E,R,B,T], [G,O,B,N,E,A,R,L,T,D]).
```

Fig. 10.15. The annotated `DONALD + GERALD = ROBERT` FD program.

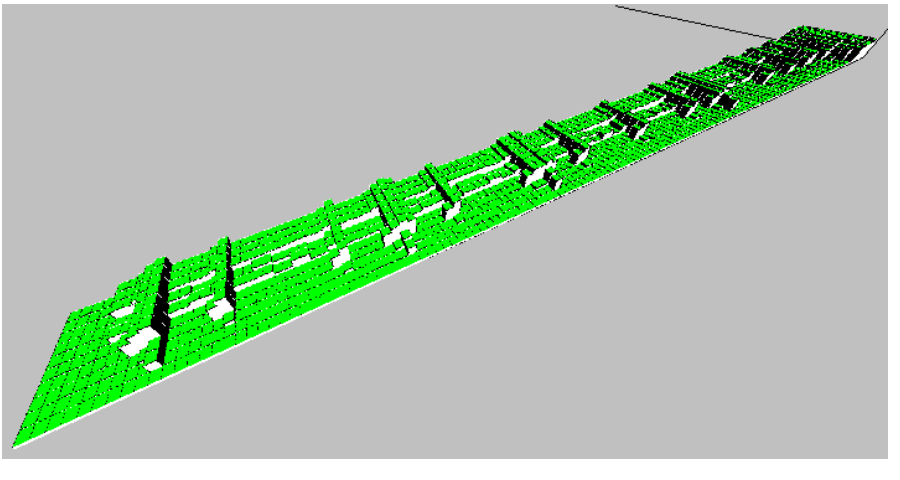


Fig. 10.16. Execution of the `DONALD + GERALD = ROBERT` program, first ordering.

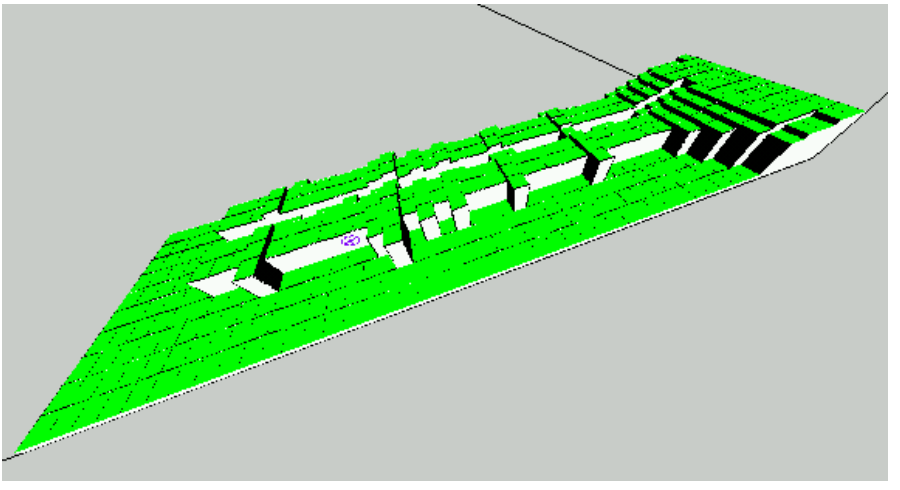


Fig. 10.17. Execution of the `DONALD + GERALD = ROBERT` program, second ordering

of the picture) which appears to be highly constrained, so that its domain is reduced right from the beginning. That variable is probably a good candidate to be labelled soon in the execution. Some other variables apparently have a high interdependence (at least, from the point of view of the solver), because in case of backtracking, the change of one of them affects the others. This suggests that the behaviour of the variables in this program can be classified into two categories: one with highly related variables (those whose domains change at once in the case of backtracking) and a second one which contains variables relatively independent from those in the first set.

Another execution of the same program, using the second ordering, yields the profile shown in Figure 10.17. Compared to the first one, there are fewer execution steps, but, of course, the classification of the variables is the same: the whole picture has the same general layout, and backtracking takes place in blocks of variables.

These figures have been generated by a tool, TRIFID, integrated into the VIFID environment. They were produced by processing a log created at runtime to create a VRML depiction with the ProVRML package [10.24], which allows reading and writing VRML code from Prolog, with a similar approach to the one used by PiLLoW [10.7]. One advantage of using VRML is that sophisticated VRML viewers are readily available for most platforms. The resulting VRML file can be loaded into such a viewer and rotated, zoomed in and out, etc. Additionally, the log file is amenable to be post-processed using a variety of tools to analyse and discover characteristics of the execution, in a similar way as in [10.16]. Another reason to use VRML is the possibility of using hyper-references to add information to the depiction of the execution without cluttering the display. In the examples shown, every variable

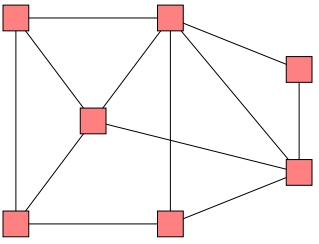


Fig. 10.18. Constraints represented as a graph

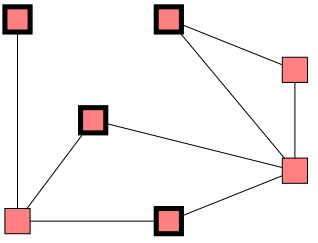


Fig. 10.19. Bold frames represent definite values

can be assigned a hyper-link pointing to a description of the variable. This description may contain pieces of information such as the source name of the variable, the actual size of its domain at that time, a profile of the changes undergone by that particular variable during the execution, the number of times its domain has been updated, the number of times backtracking has changed its domain, etc. Using the capability of VRML for sending and receiving messages, and for acting upon the receipt of a message, it is possible to encode in the VRML scene an abstraction of the propagation of constraints as it takes place in the constraint solver, in a similar way as the variable interactions depicted by VIFID.

10.4.3 Abstracting Constraints

As the number and complexity of constraints in programs grow, if we resort to visualising them as relationships among variables (e.g., 2-D or 3-D grids plus sliding bars to assign values for other variables, as suggested in Section 10.3), we may end up with the same problems we faced when trying to represent values of variables, since we are building on top of the corresponding representations. The solutions suggested for the case of representation of values are still valid (fish-eye view, abstraction of domains, ...), and can give an intuition of how a given variable relates to others. However, it is not always easy to deduce from them how variables are related to each other, due to the lack of accuracy (inherent to the abstraction process) in the representation of the variables themselves.

A different approach to abstracting the constraints in the store is to show them as a graph (see, e.g., [10.22] for a formal presentation of such a graph), where variables are represented as nodes, and nodes are linked iff the corresponding variables are related by a constraint (Figure 10.18)[4]. This representation provides the programmer with an approximate understanding of the constraints that are present in the solver (but not exactly *which* constraints

[4] This particular figure is only appropriate for binary relationships; constraints of higher arity would need hyper-graphs.

they are), after the possible partial solving and propagations performed up to that point. Moreover, since different solvers behave in different ways, this can provide hints about better ways of setting up constraints for a given program and constraint solver.

The topology of the graph can be used to decide whether a reorganisation of the program is advantageous; for example, if there are subsets of nodes in the graph with a high degree of connectivity, but those subsets are loosely connected among them, it may be worth to set up the tightly connected sections and making a (partial) enumeration early, to favour more local constraint propagation, and then link (i.e., set up constraints) the different regions, thus solving first locally as many constraints as possible. In fact, identifying sparsely connected regions can be made in an almost automatic fashion by means of clustering algorithms. For this to be useful, a means of accessing the location in the program of the variables which appears depicted in the graph is needed. This can as well help discover unwanted constraints among variables—or the lack of them.

More information can be embedded in this graph representation. For example, weights in the links can represent various metrics related to aspects of the constraint store such as the number of times there has been propagation between two variables or the number of constraints relating them. The weights themselves need not be expressed as numbers attached to the edges, but can take instead a visual form: they can be shown, for example, as different degrees of thickness or shades of colour. Variables can also have a tag attached which gives visual feedback about interesting features. For example, the actual range of the variable, or the number of constraints (if it is not clear from the number of edges departing from it) it is involved in, or the number of times its domain has been updated.

The picture displayed can be animated and change as the solver proceeds. This can reflect, for example, propagation taking place between variables, or how the variables lose their links (constraints) with other variables as they acquire a definite value. In Figure 10.19 some variables became definite, and as a result the constraints between them are not shown any more. The reason for doing so is that those constraints are not useful any longer: this reflects the idea of a system being progressively simplified. It may also help to visualise how backtracking is performed: when backtracking happens, either the links reappear (when a point where a variable became definite is backtracked over and a constraint is active again in the store), or they disappear (when the system backtracks past a point where a constraint was created).

Further filtering can be accomplished by selecting which types of constraints are to be represented (e.g, represent only “greater than” constraints, or certain constraints flagged in the program through annotations). This is quite similar to the domain filtering proposed in Section 10.4.1.

10.5 Conclusions

We have discussed techniques for visualising data evolution in CLP. The graphical representations have been chosen based on the perceived needs of a programmer trying to analyse the behaviour and characteristics of an execution. We have proposed solutions for the representation of the run-time values of the variables and of the run-time constraints. In order to be able to deal with large executions, we have also discussed abstraction techniques, including the 3-D rendition of the evolution of the domain size of the variables. The proposed visualisations for variables and constraints have been tested using two prototype tools: VIFID and TRIFID. These visualisations can be easily related, so that tools based on them can be used in a complementary way, or integrated in a larger environment. In particular, in the environment that we have developed, each tool can be used independently or they can all be triggered from a search-tree visualisation (e.g., APT).

VIFID (and, to a lesser extent, TRIFID which is less mature) has evolved into a practical tool and is are quite usable by itself as a library which can be loaded into a number of CLP systems. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed by Cosytec for the CHIP system, and which are described in other chapters.

Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments. Also thanks to Abder Aggoun and Helmut Simonis and other DiSCiPl project members for many discussions on constraint visualisation.

The implementation of VIFID was done by José Manuel Ramos [10.23]. The prototype implementation of TRIFID was done by Göran Smedbäck.

The material in this chapter comes mainly from DiSCiPl project reports, parts of which were also presented at the 1999 Practical Application of Constraint Technologies and Logic Programming Conference in London [10.24], and at the AGP'98 Joint Conference on Declarative Programming [10.8].

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

10.1 A. Aggoun and H. Simonis. Search Tree Visualization. Technical Report D.WP1.1.M1.1-2, COSYTEC, June 1997. In the ESPRIT LTR Project 22352 DiSCiPl.

10.2 K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

10.3 R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, April 1997.
10.4 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, U. of Linköping Press, May 1997.
10.5 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, U. of Linköping Press, May 1997.
10.6 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.
10.7 D. Cabeza and M. Hermenegildo. WWW Programming using Computational Logic Systems (and the PILLOW/CIAO Library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.
10.8 M. Carro and M. Hermenegildo. Some Design Issues in the Visualization of Constraint Program Execution. In *AGP'98 Joint Conference on Declarative Programming*, pages 71–86, July 1998.
10.9 The CLIP Group. Program Assertions. The CIAO System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.
10.10 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. The MIT Press, 1989.
10.11 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.
10.12 M. Ducassé. A General Query Mechanism Based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Programming Language Implementation and Logic Programming, PLILP'92*, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, 1992.
10.13 M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.
10.14 M. Fabris. CP Debugging Needs. Technical report, ICON s.r.l. ESPRIT LTR Project 22352 DiSCiPl deliverable D.WP1.1.M1.1., April 1997.
10.15 J.M. Fernández. Declarative debugging for BABEL. Master's thesis, School of Computer Science, Technical University of Madrid, October 1994.
10.16 M. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 724–734. Springer-Verlag, August 1996.
10.17 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
10.18 K. Kahn. Drawing on Napkins, Video-game Animation, and Other Ways to Program Computers. *Communications of the ACM*, 39(8):49–59, August 1996.
10.19 K. Marriot and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
10.20 M. Meier. Grace User Manual, 1996. Available at http://www.ecrc.de/eclipse/html/grace/grace.html

10.21 Sun Microsystems. Animated Sorting Algorithms, 1997. Available at http://java.sun.com/applets/

10.22 U. Montanari and F. Rossi. True-concurrency in Concurrent Constraint Programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 694–716, San Diego, USA, 1991. The MIT Press.

10.23 J.M. Ramos. VIFID: Variable Visualization for Constraint Domains. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, September 1998.

10.24 G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, April 1999.

11. Debugging Constraint Programs by Store Inspection

Frédéric Goualard and Frédéric Benhamou

IRIN, Université de Nantes
B.P. 92208, F-44322 Nantes Cedex 3
email: {goualard,benhamou}@irin.univ-nantes.fr

Expressiveness of constraint programming permits solving elegantly and efficiently many problems but makes difficult for the programmer to debug and understand the behaviour of his programs since most of the solving process (addition of new constraints and propagation of variables' domains modifications) is concealed by constraint solvers. However, tackling this problem by displaying a graphical representation of the original set of added constraints (*store*) is useless due to its huge size and its lack of structure whatsoever. We present in this chapter a means to organise hierarchically the store in order to divide it into manageable parts while preserving computation correctness; soundness of the method is shown, an algorithm supporting it is given, and an implemented prototype exhibiting its effectiveness is described.

11.1 Introduction

Constraint programming [11.5, 11.13] is a language-independent paradigm that permits solving a problem by simply specifying the properties its solution must verify. Resolving problems that way leads to programs that are both elegant and efficient. This declarativeness led many researchers to introduce it in different languages, be they imperative (REF-ARF [11.10]), object-oriented (ILOG Solver [11.21]), logic (CHIP [11.1], Prolog IV [11.3], clp(fd) [11.6]), or functional (HOPE [11.8]).

However, constraint programming (CP) is not as widely used by industry as its potential entitles it. The lack of programming environments offering advanced features for easily debugging and profiling constraint programs is certainly not extraneous to this situation. Actually, the expressiveness of constraint programming permits shortening programs at the cost of forbidding the use of traditional debuggers such as step-by-step tracers, since the main part of the solving process—that is, the addition of constraints into a constraint network called the *store* gathering all previously introduced constraints, and the re-invocation of all the constraints sharing some variables whose domain of possible values has been reduced by such additions—has no direct counterpart in the source code and is concealed from the programmer.

In this chapter, we present an abstraction of the store, called *S-box*—where the "S" stands for "store"—permitting to gather sets of constraints into one new "global" constraint that is the conjunction of the embedded

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 273–297, 2000.

constraints. The S-box abstraction allows us to device a new kind of CP debugger that discloses to the programmer the contents of the store in a usable way: a set $\mathcal{S}$ of numerous constraints having a global semantics on its own may be represented in the store by a new constraint (S-box formed by the constraints in $\mathcal{S}$), thus decreasing its complexity. Moreover, since S-boxes may contain other S-boxes, the S-box abstraction leads to an organisation of the store into a hierarchy that may be explored in a handy way by the user focusing in and out the S-boxes he wants to inspect. Properties of S-boxes are such that bugs in a constraint program may be tracked down by isolating them into smaller and smaller sets of constraints put into suitable S-boxes. The declarative semantics of the program is shown not to be affected by this debugging process though the re-invocation order of constraints is modified.

The rest of the chapter is organised as follows: Section 11.2 briefly introduces some fundamental constraint programming notions, focusing on *constraint logic programming*, that is CP embedded in a logic language such as Prolog [11.7]; Section 11.3 describes the difficulties the programmer has to face when debugging constraint programs with traditional tools; Section 11.4 introduces the S-box notion permitting to structure hierarchically the constraint store; its soundness is shown, and an algorithm supporting S-boxes is given to replace the propagation algorithms of constraint solvers; Section 11.5 presents the functionalities of a prototype for a S-box-based debugger, while Section 11.6 describes its implementation details. Last, Section 11.7 summarizes the contribution and points out some directions for future researches.

11.2 Constraint Logic Programming in a Nutshell

For the sake of simplicity, we will restrict ourselves in the following to *Constraint Logic Programming* (CLP) and to numerical constraints involving only integer-valued variables.

Consider the constraint $c_1 : x + y = z$ where the initial domains for the variables are respectively $D_x = \{2, 3, \ldots, 6\}$, $D_y = \{2, \ldots, 4\}$, and $D_z = \{3, \ldots, 5\}$. Some values of these domains are clearly inconsistent with c_1. For example, x cannot take the value 6 since there is no value in D_y and D_z such that $6 + y = z$. Hence, this value may be eliminated from D_x. A *constraint narrowing operator* [11.2] $N[c]$ for a n-ary constraint c is a function that discards the n-tuples from a Cartesian product of domains that are inconsistent for c. Given two Cartesian products of domains $\boldsymbol{D}$ and $\boldsymbol{D'}$, a contracting operator $N[c]$ for the constraint c (with an underlying relation ρ) must have the following properties:

$$
\begin{array}{ll}
N[c](\boldsymbol{D}) \subseteq \boldsymbol{D} & \text{contractance} \\
\quad \rho \cap \boldsymbol{D} \subseteq N[c](\boldsymbol{D}) & \text{correctness} \\
\boldsymbol{D} \subseteq \boldsymbol{D'} \Rightarrow N[c](\boldsymbol{D}) \subseteq N[c](\boldsymbol{D'}) & \text{monotonicity}
\end{array}
$$

The filtering of consistent values is related to *local consistency* notions [11.15, 11.18]. In the example given above, applying the $N[c_1]$ operator to the Cartesian product $D_x \times D_y \times D_z$ would narrow the domains to $D'_x = \{2, 3\}$, $D'_y = \{2, 3\}$, and $D'_z = \{4, 5\}$. The remaining values are such that for every α in D'_x (resp. D'_y and D'_z), there exists at least one pair (β, γ) of consistent values in $D'_y \times D'_z$ (resp. $D'_x \times D'_z$ and $D'_x \times D'_y$) such that $c_1(\alpha, \beta, \gamma)$ does hold (*arc consistency* [11.15]). If y is later bound to 3 (e.g., by adding the constraint $c_2 : y \geqslant 3$), the $N[c_1]$ operator will be able to further narrow down D'_x and D'_z to the final domains $D''_x = \{2\}$, $D''_z = \{5\}$. Consequently, constraints sharing variables have to be "linked" in some way, thereby permitting the re-invocation of all the constraints where occurs a variable whose domain has just been narrowed by one of them. In this chapter, a *constraint store* is defined as such a network where nodes are constraints linked together whenever they share at least one variable (see example in Figure 11.1).

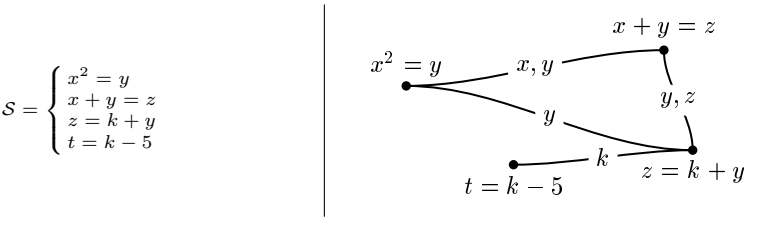


Fig. 11.1. A constraint system and its associated store

Algorithm 11.1 describes in details the addition of constraints into the store: constraints $c_1, \ldots, c_m$ are added into a *propagation queue* (managed as a FIFO list in most implementations); their narrowing operators $N[c_1], \ldots, N[c_m]$ are applied onto the Cartesian product of the variables' domains, and all the constraints involving a variable whose domain has been narrowed are pushed into the propagation queue. The whole process terminates on failure (i.e. a domain is narrowed to the empty set) or on emptiness of the propagation queue (the greatest common fixed-point of all the narrowing operators is reached).

Narrowing operators are in general unable to discard all the inconsistent values from the Cartesian product of domains (*partial consistency* only). Hence, applying Algorithm 11.1 is often insufficient to achieve the overall consistency. Then, a *labelling process*—consisting in assigning to each variable all of its possible remaining values through *backtracking*—occurs when all constraints have been added to the store, thus permitting to isolate all the consistent assignments for the variables involved in the constraint system.

Algorithm 11.1. NAR: the narrowing algorithm

```
NAR(in: {N[c_1], ..., N[c_m]};
    inout: D = D_1 × ··· × D_n)   % n: total number of variables,
                                  % with cylindrification of constraints
begin
  Q := {c_1, ..., c_m}   % Adding c_is in the propagation list
  S := S ∪ {c_1 ..., c_m}   % Adding c_is in the store
  while (Q ≠ ∅ and D ≠ ∅) do
    c := choose a constraint c_i in Q
    D' := N[c_i](D)
    if (D' ≠ D) then
      Q := Q ∪ {c_j ∈ S | ∃x_k ∈ Var(c_j) ∧ D'_k ≠ D_k}
      D := D'
    endif
    Q := Q \ {c}
  endwhile
end.
```

11.3 Arising Difficulties when Debugging Constraint Programs

In this section, an example demonstrating the efficiency of the constraint programming paradigm in comparison with a naive imperative solution is shown, and the drawbacks of the qualities permitting such an efficiency with respect to debugging are pointed out.

11.3.1 Constraint Programming Efficiency

Consider the following problem solved thereafter in both an imperative language and a constraint programming language:

Example 11.3.1 (Yoshigahara problem [11.23]). Given nine integer-valued variables $A, \ldots, I$, use all digits 1 through 9 only once such that

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \tag{11.1}$$

where BC, EF and HI stand for the number obtained by concatenating the digit values of the corresponding variables.

The Yoshigahara problem has only one solution (up to permutations of quotients): $5/34 + 7/68 + 9/12 = 1$.

A naive solution for this problem involves the generation of the $9^9 = 387\,420\,489$ possible assignments to variables A through I, followed by the

test whether each assignment meets the requirement (11.1). This is the so-called *generate and test* approach illustrated by the Java program given in Program 11.1.

Program 11.1. The Yoshigahara problem solved in Java

```
class Yoshigahara {
 private static boolean alldifferent(int a, int b, ...) {
  return (a!=b && a!=c && a!=d && ...
          && f!=i && g!=h && g!=i && h!=i);
 }
 public static void main(String[] args) {
  for (int A=1;A<=9;A++)                                  // Generate
   ...
    for (int I=1;I<=9;I++) {
     if (alldifferent(A,B,C,D,E,F,G,H,I)) {                // Test
      float res=A/(10.f*B + C) + D/(10.f*E + F)+
                G/(10.f*H + I);
      if (res == 1.0)
       System.out.println(A+"/"+B+C+" + "+D+"/"+E+F+
                          " + "+G+"/"+H+I+" = 1");
     }
    }
  }
}
```

On the other hand, a constraint-based resolution starts stating all the constraints before generating remaining possible assignments. As a result, many assignments are not considered. For example, if the addition of a constraint leads to discarding the value 6 from the domain of the A variable, all the 9-tuples of the form $(6, \_, \_, \_, \_, \_, \_, \_, \_)$ are no longer considered during the labelling phase. This is the *test and generate* approach illustrated by the DecLIC [11.11] program given in Program 11.2.

Program 11.2. The Yoshigahara problem solved in DecLIC

```
:- L=[A,B,C,D,E,F,G,H,I],
   is_integer(L),                                          % Test
   alldifferent(L),
   domain(L,1,9),
   A/(10*B+C)+D/(10*E+F)+G/(10*H+I) $= 1,
   labelingff(L).          % Generate/enumerate (smallest domain first)
```

On an AMD K6-166 MHz based PC, all solutions are found by Program 11.1 in 1 778.2s. and by Program 11.2 in 1.83s.

11.3.2 Drawbacks of CP Expressiveness

Debugging Program 11.1 is easily done using traditional debuggers allowing to run it step-by-step since every instruction performed to solve the problem appears in the code. This is not the case for Program 11.2. The point is that the actual work is not reflected by the instructions appearing in it. For example, if we use a Prolog tracer to run it step-by-step from Line 4 to Line 6, we will only be able to see the domain modifications induced by introducing the constraint appearing in Line 5 into the store, and we will miss everything that was done when propagating intermediate modifications in the constraint network (reinvocation of constraints and intermediate modifications of domains). In particular, we get no information at all in case of failure.

Actually, there exist two independent levels in CLP programs:

1. the Prolog level (clausal structure) which may be debugged using the traditional debugging facilities;
2. and the store level (propagation of domain modifications) for which traditional debugging facilities are useless.

The store level is clearly unreachable from traditional debuggers. Thence, a specialized tool, able to display the constraint store in a readable way, has to be devised for this purpose.

11.4 Visualising the Store

The debugging tools integrated in CP languages such as ECL$^{\text{i}}$PS$^{\text{e}}$ [11.17] and Oz [11.22] do not allow to visualise directly the store, but display the variable domains and their modifications "in real time". Though these are important and useful functionalities, we believe that a more direct presentation of the relations between the constraints in the store is mandatory to fully debug CP programs. It sounds particularly true when variables are real-valued, since then, domain modifications bring few information.

The idea is then to display a representation of the store as a constraint network such as the one appearing in Figure 11.1. However, displaying the store as it is manipulated by the solver (raw form) is useless since it is a huge, flat, collection of constraints with no structure at all—even for toy problems. Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (e.g. in Prolog IV), and stored constraints are not always those given by the user (e.g. solvers like clp(BNR) decompose constraints into some primitive constraints, using new variables).

It is then necessary to let the user control the size and complexity of the presentation of the store.

11.4.1 Structuring the Store

A CLP program is structured by clauses. This structure is lost when the constraints are added to the store whereas it often conveys some useful information concerning the meaning of some constraint sets. For example, let us consider Program P [11.12] (Prog. 11.3) describing the problem of finding the collision point between a wall and a ball.

Program 11.3. Code for Program P with tagged Prolog goals

```
object_A(X,Y,Z):-                                  % Shape of the wall
        X ⩽ 0, Y ⩽ 0, Z ⩽ 0.
object_B(XCx,YCy,ZCz):-                            % Shape of the ball
        XCx^2 + YCy^2 + ZCz^2 = 1.
center_B(T,Cx,Cy,Cz):-                             % Pos. of the ball center at T
        T^2 - Cx = 10,
        2T - Cy = 10,
        Cz - T^2 + 7T = 10.
object_B_moving(T,X,Y,Z):-                         % Pos. of pt. (X,Y,Z) in ball at T
        center_B(T,Cx,Cy,Cz),
        XCx = X - Cx,
        YCy = Y - Cy,
        ZCz = Z - Cz,
        object_B(XCx,YCy,ZCz).
:- T ⩾ 0, object_A(X,Y,Z), object_B_moving(T,X,Y,Z).
```

The `center_B(`T`,`Cx`,`Cy`,`Cz`)` clause, defining the center position of the ball, contains three constraints. It could be considered as a new user-defined "global constraint" whose semantics is quite intuitive. However, introducing these three constraints in the store discards the special link between them. Figure 11.2 displays the store for Program P.

Note that it is already an abstraction of the actual store since all the constraints appearing in it are the constraints as given by the user (no decomposition into primitives, nor renaming of variables as usually done in Prolog); and yet, it may already be too complicated to be useful.

A first approach to reduce the store complexity is to reintroduce the structure induced by the program clauses. Figure 11.3 shows the store for P where each clause is represented by a box (more precisely, the behaviour at runtime is as follows: a box is created whenever the control "enters" into a clause; added constraints are then put into that box until the control leaves the clause).

Structuring the store that way permits presenting abstractions of it by masking all the constraints in a box at some level. For example, the store of Figure 11.3 may be abstracted into an equivalent store containing only three

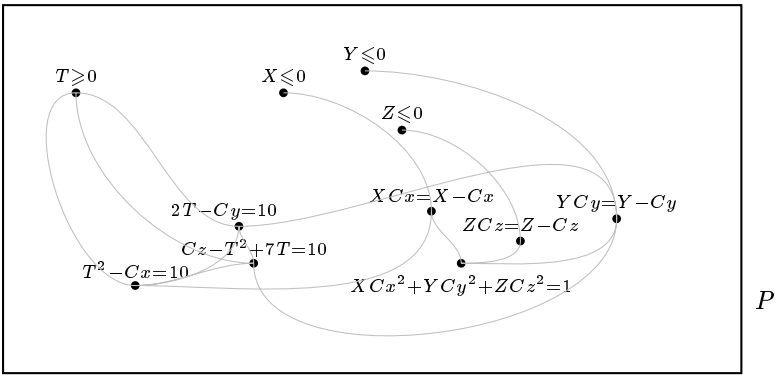


Fig. 11.2. Store for P with programmer's constraints

constraints: $T \geqslant 0$, `object_A`(X,Y,Z), and `object_B_moving`(T,X,Y,Z)—Figure 11.4.

Note that the number of edges between "constraints" has dramatically decreased. It is then easier to add some guards onto these links warning the programmer of the violation of some property when propagating variable domains from a S-box to another. For example, assume that the propagation of some domains given by the user as an input to the box `object_B_moving`(T, X, Y, Z) does no lead to the right domains on output; in order to find what went wrong, the user can focus on the S-box of `object_B_moving`(T, X, Y, Z), obtaining the store described in Figure 11.5.

Focusing on a S-box σ means temporarily restricting the set of constraints in the store to the ones contained in σ and the sub-S-boxes it embeds. As

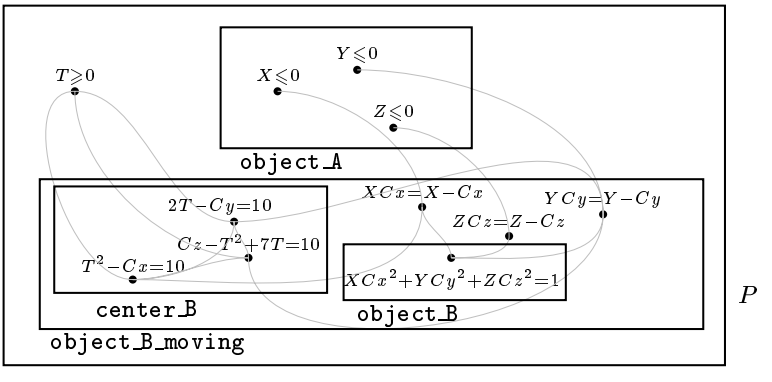


Fig. 11.3. Structure of the store for P (all goals tagged)

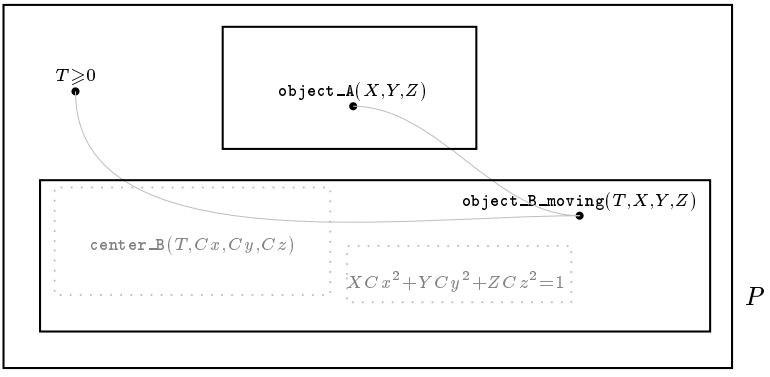


Fig. 11.4. Abstraction of the store for P

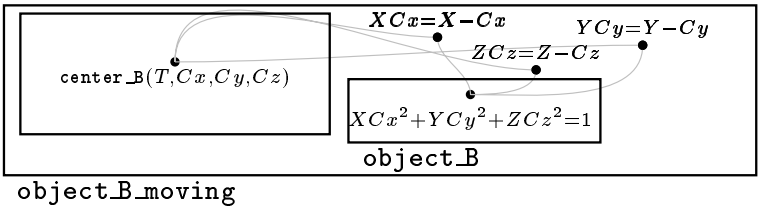


Fig. 11.5. The new store after focusing on `object_B_moving`(X, Y, Z)

a consequence, propagation of domains' narrowing has to be done only in the S-box having the *focus*—otherwise, we cannot be assured that the bug is really in the box—and the focus must be reset to the entire store sooner or later during the debugging process, such that all constraints have a chance to be reconsidered (preservation of the program declarative semantics).

A greater flexibility may be obtained by letting the programmer create boxes which are not directly inspired by the clausal structure of the program. This can be done by tagging all the constraints to be put in the same box.

11.4.2 S-Boxes

The "box" notion discussed previously is now formally defined in this section.

Let us consider the constraint system $\mathcal{S} = \{c_1, \ldots, c_m\}$ along with the Cartesian product of the domains of the variables occurring in constraints of $\mathcal{S}$, $\boldsymbol{D} = D_1 \times \cdots \times D_n$. The declarative semantics $\mathcal{S}^\star$ of $\mathcal{S}$ corresponds to the set of n-tuples of $\boldsymbol{D}$ satisfying the conjunction $c_1 \wedge \cdots \wedge c_m$. In general, one must content oneself with an approximation $\overline{\mathcal{S}}$ of this semantics (e.g. case of

real-valued variables). This approximation is related to the greatest common fixed-point of the $N[c_i]$ included in $\boldsymbol{D}$ [11.2][1]:

$$\overline{\mathcal{S}} = \max(\{u \in \bigcap_{i=1}^{m} \text{fixed-point}(N[c_i]) \mid u \subseteq \boldsymbol{D}\})$$

Let $C = c_1 \wedge \cdots \wedge c_m$ be the constraint whose narrowing operator $N[C]$ computes $\overline{\mathcal{S}}$. Then, the constraint set $\{c_1, \ldots, c_m\}$ may be replaced in any store by C without modifying the program declarative semantics.

Constraint C may in the same way be integrated in another constraint C'. A huge set of constraints may thus be organised into a hierarchy, preserving its semantics. This idea is at the root of the *S-box notion*:

Definition 11.4.1 (S-box). *Let $\mathcal{V}$ be a set of n variables, $\boldsymbol{D} = D_1 \times \cdots \times D_n$ a Cartesian product of domains, and $\mathcal{C}$ a set of constraints. A* S-box *is a non-empty set $\sigma = \{a_1, \ldots, a_m\}$, where a_i is either a constraint from $\mathcal{C}$ or a S-box. The narrowing operator $N[\sigma]$ associated to the S-box σ is defined as:*

$$N[\sigma]\colon \boldsymbol{D} \to \boldsymbol{D}$$
$$X \mapsto N[\sigma](X) = \max(\{u \in \bigcap_{i=1}^{m} \text{fixed-point}(N[a_i]) \mid u \subseteq X\})$$

The operator associated to Algorithm NAR (Alg. 11.1, p. 276) has the following properties [11.19]: it is *contracting, correct, confluent*, and *monotone*. Hence, it is a narrowing operator. Since we have also $\text{NAR}(\mathcal{S}, \{D_1 \times \cdots \times D_n\}) = \overline{\mathcal{S}}$ for $\mathcal{S} = \{c_1, \ldots, c_m\}$, we deduce that NAR may be used in practice as an implementation of the narrowing operator $N[\sigma]$ of an S-box σ.

However, in order to fulfil the "box notion" given in the previous section (i.e. a set of constraints in a box is considered as a new constraint on its own), the narrowing operator $N[\sigma]$ has to be atomically computed, that is, the domains' propagation fixed-point must be reached in each S-box σ of a S-box hierarchy without re-invoking any constraint outside σ in between. For example, consider the hierarchy corresponding to the store of Figure 11.3 (see tree in Figure 11.6): operationally, assuming that each S-box owns its own propagation queue in which are pushed the constraints to re-invoke belonging to that S-box, it is forbidden to consider the propagation queue of one S-box "above" or at the same level as the current one if all the S-box queues "below" it (including its own) are not empty. The difficulty arises from the fact that considering a constraint out of one queue may add a constraint to any queue in the tree. For example, assume that the current S-box is ③. Then, popping constraints out of its queue may add constraints into the queues of ④ and ⑤.

[1] We consider the greatest fixed-point since we want to preserve correctness (all solutions kept).

It is then necessary to consider these queues before considering ① and ②. However, processing ⑤ after having considered ④ may add new constraints to the queue of ④ such that it is necessary to return processing ④. This process will eventually finish since all operators are contracting ones over a finite set of numbers. Nevertheless, we have to traverse the tree following a complicated path before reaching quiescence.

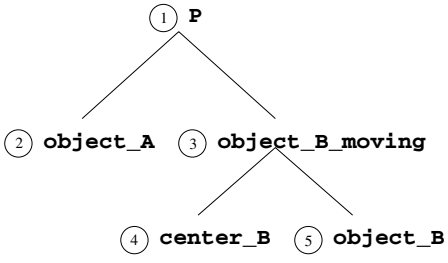


Fig. 11.6. Tree-structured representation of the S-box hierarchy of Figure 11.3

The idea is then to use only one propagation queue and to rely on the Dewey notation properties [11.14]. Given a S-box hierarchy represented as a tree, we associate to each constraint the Dewey index of its S-box in the tree-structured representation (see Figure 11.7); during propagation, each constraint c' to be reconsidered is added in the propagation list at a position such that the list is sorted according to the order $\mathcal{O}_c$ defined below (with c being the last reinvoked constraint before adding c').

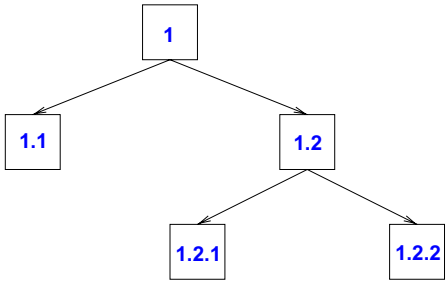


Fig. 11.7. S-box hierarchy of Figure 11.6 with the Dewey notation

Notations. Let P be a constraint program, and σ a S-box containing a constraint c of P. The $\mathcal{I}_c$ index of c corresponds to the Dewey index of σ in the S-box hierarchy of P. Let $\mathsf{SzComPref}(\mathcal{I}_1,\mathcal{I}_2)$ be the function returning the size (number of signs) of the common prefix of indices $\mathcal{I}_1$ and $\mathcal{I}_2$.

Definition 11.4.2 (Partial constraint ordering $\mathcal{O}_c$). *Let c_1, c_2, and c be constraints. Constraint c_1 is said smaller than c_2 with respect to c ($c_1 \overset{c}{\prec} c_2$) if and only if:*

$$c_1 \overset{c}{\prec} c_2 \iff \begin{cases} 1.\ \mathsf{SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) > \mathsf{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \\ 2.\ \textit{or } \mathsf{SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) = \mathsf{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \\ \quad \textit{and } \mathcal{I}_{c_2} >^{lexe} \mathcal{I}_{c_1} \\ \quad\quad (>^{lexe}\textit{: lexicographical extension of } > \textit{ over naturals}) \end{cases}$$

Note that two constraints with the same index (that is, in the same S-box) are unordered with respect to any constraint.

Example 11.4.1 (Partial ordering on constraints). Let c_1, c_2, c_3, and c_4, be constraints with indices $\mathcal{I}_{c_1} = 1.1.5$, $\mathcal{I}_{c_2} = 1.1.4$, $\mathcal{I}_{c_3} = 1.1.5.6$, and $\mathcal{I}_{c_4} = 3.1.4$. We have the following inequalities:

$$\begin{cases} c_1 \overset{c_3}{\prec} c_2 \\ c_2 \overset{c_4}{\prec} c_1 \end{cases}$$

The propagation strategy based on sorting the propagation queue with respect to the last popped constraint may then be proved to ensure atomicity when computing the fixed-point in the S-boxes by noting that Rule 1 ensures that constraints in the S-boxes closest to the S-box of c (last reinvoked constraint) are put at the head of the propagation queue (locality principle), and Rule 2 sorts constraints in the S-boxes "above" or at the same level than σ in such a way that the locality principle be respected for them when they are later considered.

Algorithm NAR needs to be slightly modified in order to take into account the new propagation scheme. The resulting algorithm called REVINCNAR (see Algorithm 11.2) is an incremental modified version of NAR supporting the S-box abstraction.

Let $c_1, \ldots, c_m$, be the constraints appearing in the propagation list $\mathcal{Q}$ (in that order). Functions used in Algorithm 11.2 have the following meaning:

SBOX(c) : returns the S-box containing c;
DEWEY(c) : returns the Dewey index of Constraint c;
INSERT$_\iota(\mathcal{Q}, c)$: inserts the constraint c in the propagation queue $\mathcal{Q}$ just before the constraint c_i such that:

$$\begin{cases} \forall j \in \{1, \ldots, i-1\} : \mathcal{I}_{c_j} \overset{\iota}{\lhd} \mathcal{I}_c \\ \forall j \in \{i, \ldots, m\} : \mathcal{I}_{c_j} \overset{\iota}{\trianglerighteq} \mathcal{I}_c \end{cases}$$

where $\lhd$ is the ordering on Dewey indices corresponding to the ordering $\prec$ on constraints, and ι is the Dewey index of the S-box containing the constraint being reinvoked;

UnderFocus(σ) : returns true if the Dewey index of the S-box σ_f having the focus is a prefix of the Dewey index of σ (i.e. σ is a "descendant" of σ_f);
Top($\mathcal{Q}$) : returns c_1 from $\mathcal{Q}$. Note: the constraint is not removed from $\mathcal{Q}$;
PopHead($\mathcal{Q}$) : discards c_1 from $\mathcal{Q}$.

Algorithm 11.2. RevIncNar: Nar revisited to support S-boxes

```
RevIncNar(in: N[c1] ; inout: D = D1 × ··· × Dn)
begin
  Q:= Insert_ι(Q, c1)   % Constraint added to the propagation list
  S:= S ∪ {c1}          % Constraint added to the store
  while not(Empty(Q)) and UnderFocus(SBox(Top(Q)))
          and D ≠ ∅ do
        (Q,c) := PopHead(Q)
        ι := Dewey(c)
        D' := N[c](D)
        if (D' ≠ D) then
              forall {cj ∈ S | ∃xk ∈ Var(()cj) ∧ D'k ≠ Dk} do
                    Q:= Insert_ι(Q, cj)
              endforall
              D := D'
        endif
  endwhile
end
```

Note that Index ι has to be initialised to 1 (where "1" is the Dewey index of the S-box containing all the constraints of the debugged program) somewhere before the first call to RevIncNar. Note also that $\mathcal{Q}$ may be non-empty when entering or leaving RevIncNar. This is the case, for example, whenever the focus is not set to the whole store and there exists in $\mathcal{Q}$ some constraints which are not under the focus.

11.5 Presentation of the Debugger Prototype

A prototype for a S-box based debugger has been implemented over clp(fd) [11.6], an extension of Prolog by Codognet and Diaz for constraint logic programming with integer-valued variables. Functionalities of the prototype are briefly surveyed here.

Figure 11.8 displays a snapshot of the debugger execution on `cars`, a sequencing problem extracted from the examples of the clp(fd) distribution (see code in Prog. 11.4).

The main window is composed of two parts:

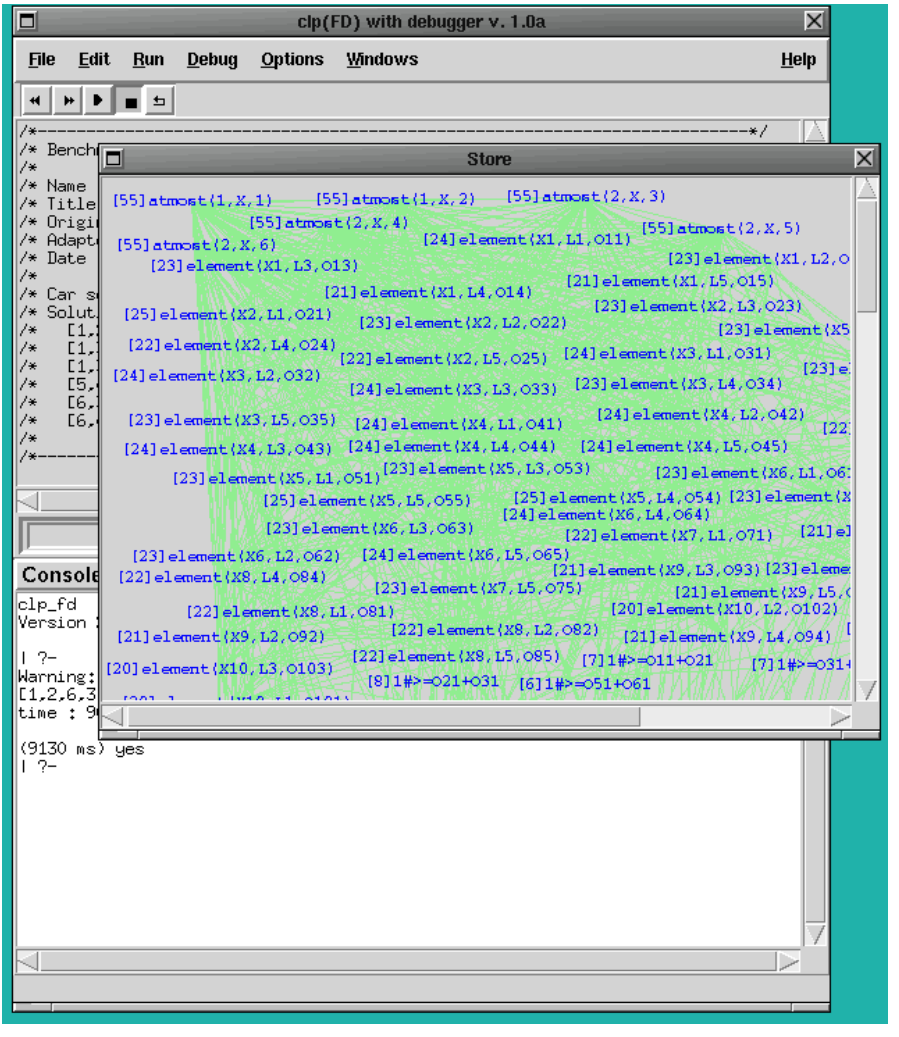


Fig. 11.8. Debugging `cars.pl` without S-boxes

1. an editor containing the source code to interpret;
2. a console for the clp(fd) system where goals may be written interactively.

S-box creation may be done in the editor by tagging with the mouse some sets of constraints and/or goals, and giving them a name.

The debugger permits visualising graphically the variable domain modifications. Constraints and variables appear in the S-boxes with the same aspect as in the source code (e.g. variable names are preserved, which is not the case in many Prolog debuggers).

Program 11.4. clp(fd) code for `cars.pl`

```
:- cars(L).
cars(X):- X=[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10],
        Y=[O11,O12,O13,O14,O15,O21,O22,O23,O24,O25,O31,O32,O33,O34,O35,O41,O42,O43,O44,O45,O51,O52,O53,
        O54,O55,O61,O62,O63,O64,O65,O71,O72,O73,O74,O75,O81,O82,O83,O84,O85,O91,O92,O93,O94,O95,
        O101,O102,O103,O104,O105],
        L1=[1,0,0,0,1,1], L2=[0,0,1,1,0,1], L3=[1,0,0,0,1,0], L4=[1,1,0,1,0,0], L5=[0,0,1,0,0,0],
        domain(Y,0,1), domain(X,1,6),
        atmost(1,X,1), atmost(1,X,2), atmost(2,X,3), atmost(2,X,4),
        atmost(2,X,5),atmost(2,X,6),
        element(X1,L1,O11), element(X1,L2,O12), element(X1,L3,O13), element(X1,L4,O14), element(X1,L5,O15),
        element(X2,L1,O21), element(X2,L2,O22), element(X2,L3,O23), element(X2,L4,O24), element(X2,L5,O25),
        element(X3,L1,O31), element(X3,L2,O32), element(X3,L3,O33), element(X3,L4,O34), element(X3,L5,O35),
        element(X4,L1,O41), element(X4,L2,O42), element(X4,L3,O43), element(X4,L4,O44), element(X4,L5,O45),
        element(X5,L1,O51), element(X5,L2,O52), element(X5,L3,O53), element(X5,L4,O54), element(X5,L5,O55),
        element(X6,L1,O61), element(X6,L2,O62), element(X6,L3,O63), element(X6,L4,O64), element(X6,L5,O65),
        element(X7,L1,O71), element(X7,L2,O72), element(X7,L3,O73), element(X7,L4,O74), element(X7,L5,O75),
        element(X8,L1,O81), element(X8,L2,O82), element(X8,L3,O83), element(X8,L4,O84), element(X8,L5,O85),
        element(X9,L1,O91), element(X9,L2,O92), element(X9,L3,O93), element(X9,L4,O94), element(X9,L5,O95),
        element(X10,L1,O101), element(X10,L2,O102), element(X10,L3,O103), element(X10,L4,O104),
        element(X10,L5,O105),
        1 #>= O11+O21, 1 #>= O21+O31, 1 #>= O31+O41, 1 #>= O41+O51, 1 #>= O51+O61, 1 #>= O61+O71,
        1 #>= O71+O81, 1 #>= O81+O91, 1 #>= O91+O101, 2 #>= O12+O22+O32, 2 #>= O22+O32+O42,
        2 #>= O32+O42+O52, 2 #>= O42+O52+O62, 2 #>= O52+O62+O72, 2 #>= O62+O72+O82,
        2 #>= O72+O82+O92, 2 #>= O82+O92+O102, 1 #>= O13+O23+O33, 1 #>= O23+O33+O43,
        1 #>= O33+O43+O53, 1 #>= O43+O53+O63, 1 #>= O53+O63+O73, 1 #>= O63+O73+O83,
        1 #>= O73+O83+O93, 1 #>= O83+O93+O103, 2 #>= O14+O24+O34+O44+O54,
        2 #>= O24+O34+O44+O54+O64, 2 #>= O34+O44+O54+O64+O74,
        2 #>= O44+O54+O64+O74+O84, 2 #>= O54+O64+O74+O84+O94,
        2 #>= O64+O74+O84+O94+O104, 1 #>= O15+O25+O35+O45+O55,
        1 #>= O25+O35+O45+O55+O65, 1 #>= O35+O45+O55+O65+O75,
        1 #>= O45+O55+O65+O75+O85, 1 #>= O55+O65+O75+O85+O95,
        1 #>= O65+O75+O85+O95+O105,
                                                                        % Redundant constraints
        O11+O21+O31+O41+O51+O61+O71+O81 #>= 4, O11+O21+O31+O41+O51+O61 #>= 3,
        O11+O21+O31+O41                 #>= 2, O11+O21                 #>= 1,

        O12+O22+O32+O42+O52+O62+O72 #>= 4, O12+O22+O32+O42             #>= 2,
        O12                         #>= 0, O13+O23+O33+O43+O53+O63+O73 #>= 2,
        O13+O23+O33+O43             #>= 1, O13                         #>= 0,
        O14+O24+O34+O44+O54         #>= 2, O15+O25+O35+O45+O55         #>= 1.
```

A window is associated to each S-box (cf. Figure 11.9). Any of them may be masked or displayed at the user will. A S-box appears like an ordinary constraint in the window of its ancestor. Fig. 11.9 presents the execution of `cars` where all the constraints of the same "form" have been put in the same S-box, and where redundant constraints have been isolated in one S-box.

Breakpoints may be added to variables (break when a domain is equal or different to another, reduced to one value, etc.), constraints (break on re-invocation), S-boxes (break on entering/leaving), and other events such as failure or entering in the labelling phase. The propagation process may be done step-by-step, allowing the programmer to understand the relations between sets of constraints/S-boxes.

11.6 Implementation of the S-Box Based Debugger

We now present the implementation details of the prototype for the S-box based constraint debugger whose functionalities have been described in the previous section.

Though efficiency is not the primary concern for a debugger, a quick handling of the user interactions is an important factor for the ease of use of

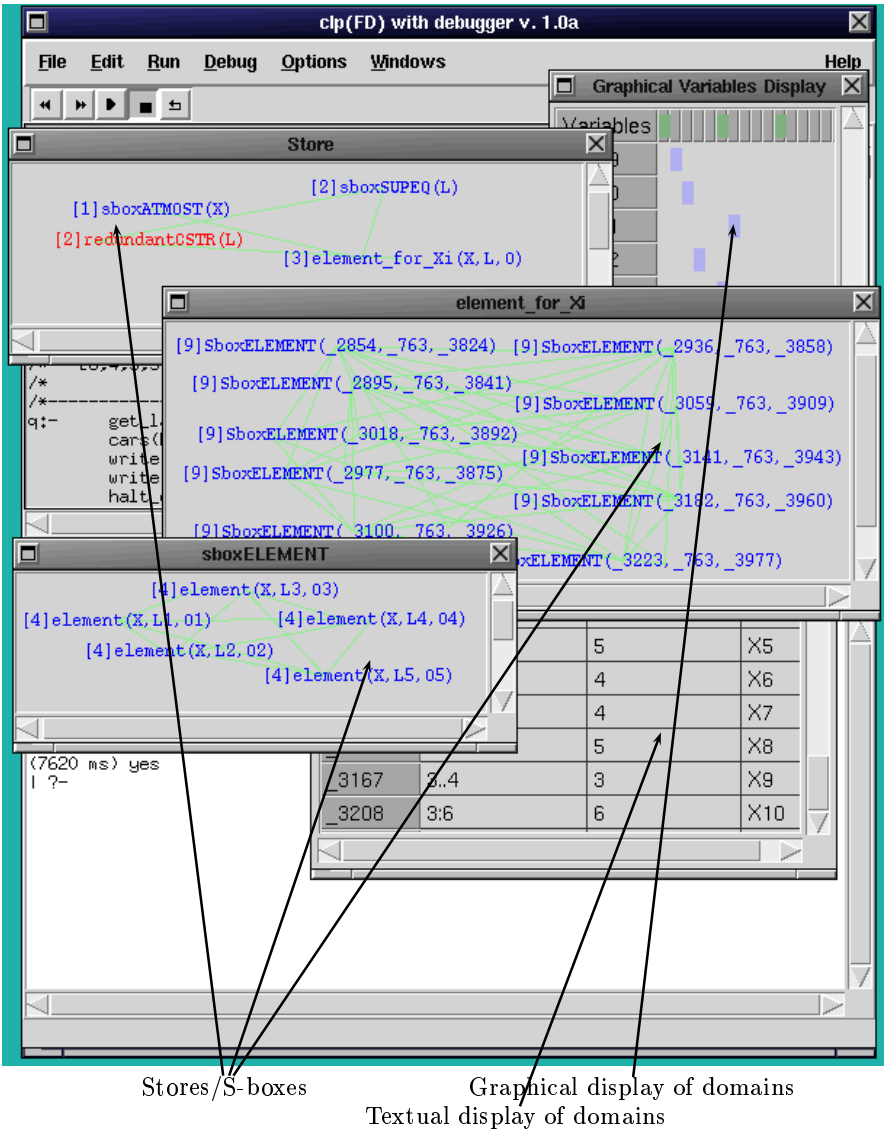


Fig. 11.9. Debugging `cars.pl` with S-boxes

the graphical user interface. This point is in favour of a tight integration of the debugger into the solver in order to minimise the overhead induced by the necessary communications between them.

On the other hand, a fully integrated debugger cannot be used with another solver without rewriting it entirely.

In order to preserve portability while avoiding the overhead of the communications between a debugger completely separated from the solver, our debugger has been devised in two parts (see Figure 11.10):

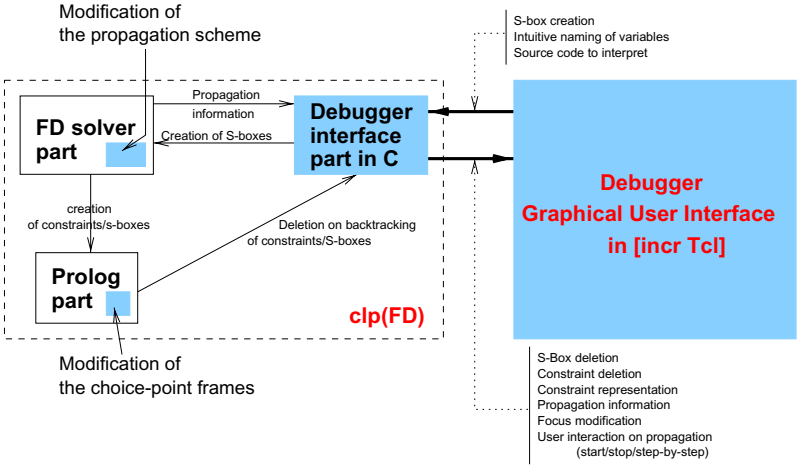


Fig. 11.10. The debugger architecture

1. a tightly integrated part written in C interacting at a low level with the solver; it implements some primitives such as methods specifying how a constraint must be displayed, and some structures entirely devoted to debugging (e.g. everything related to the S-box data);
2. a part written in [incr Tcl] [11.16] (object-oriented version of Tcl/Tk [11.20]) implementing all the graphical user interface. This part has its own representation of the structures related to the store and the S-boxes in order to minimise as much as possible the (costly) message passing between C and [incr Tcl].

The use of Tcl/Tk and [incr Tcl]—both available on a wide variety of platforms—allows us to preserve portability for the prototype. As a consequence, the debugger is portable on all the platforms already supported by clp(fd). In addition, we also support SPARC machines under Solaris.

Filled frames in Figure 11.10 represent the parts which have been added or modified in clp(fd). As one may see, the separation between the solver and the debugger is not as clean as what one would like it to be:

– the choice-point frame structures of the Prolog engine have been modified in order to keep trace of the constraints and S-boxes added to the store. This permits erasing them from the graphical representation in the gra-

phical user interface on backtracking. Figure 11.11 presents the resulting modified choice-point frame;
– the clp(fd) propagation process has been replaced by the one described above by RevIncNar.

Actually, it would be very hard to devise a S-box based debugger without achieving these modifications of the kernel. The only alternative would be to integrate the debugger in a system providing the necessary handlers to obtain the same functionalities.

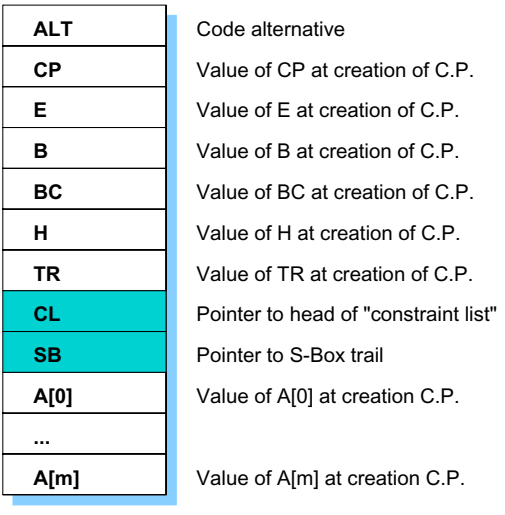


Fig. 11.11. Modified choice-point frame

Table 11.1 gathers all the registers added to the ones already present in clp(fd). Their use is presented in the following sections.

Table 11.1. Registers added to clp(fd)

name	meaning	description
SBF	*S-Box Focus*	pointer to the S-box having the focus
SBC	*S-Box Current*	pointer to the current S-box i.e. the last one created but not finished
PLH	*Propagation List Head*	—

11.6.1 Modification of the Backtracking Process

A trailing structure (see Figure 11.12) is created each time the Prolog engine creates a new choice-point. Actually, the information related to a S-box cannot be put in the Prolog trail since the S-boxes may not follow the clausal structure of the program (a S-box might be created in a clause and closed in another—see Program 11.5).

Program 11.5. S-boxes creation *vs.* clausal structure

```
1    p:- begin_sbox, q, r.                    % S-box 1.1 created
2    q:- t, end_sbox, s.                      % S-box 1.1 closed
3    t:- begin_sbox, s, end_sbox.  % S-box 1.1.1 created and closed
```

As a consequence, all the S-box-related structures needing to be reset on backtracking are saved in a local specialised trail whose address is kept in the choice-point frame.

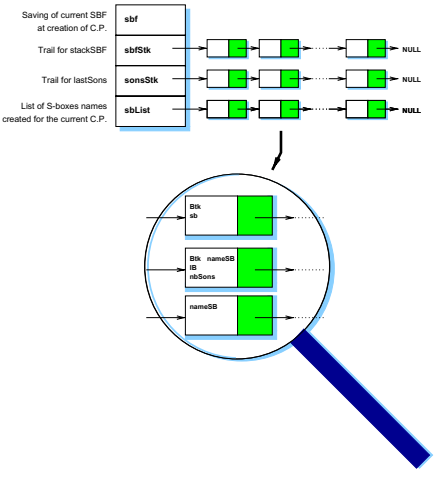


Fig. 11.12. S-box trail frame

11.6.2 Modification of the Propagation Process

In clp(fd) [11.9], the propagation of domain modification is done as follows: each time the domain of a variable v is modified, the variable frame of v is put at the end of a propagation queue (handled as a FIFO list). Relaxation

is achieved by popping the variable frame at the head of the queue and by re-invoking all the constraints whose addresses are kept in the frame (these are the constraints involving an occurrence of the variable). These re-invocations may add new variable frames at the end of the queue. Propagation ends when the queue is empty. Here, the propagation queue is "*variable-oriented*" since it is a queue of variables.

On the other hand, Algorithm 11.2 presented above uses a "*constraint-oriented*" view where the propagation queue contains only constraints. As a consequence, all the propagation process needs to be modified in order to support S-boxes. Four fields have then been added to the original constraint frame (see the resulting frame in Figure 11.13): the propagation queue is implemented by linking together the constraint frames of the constraints to re-invoke by using the field `Next_In_Queue`. The head of the propagation queue is always pointed by the register PLH. Note that we only need to know the position of the head since the introduction of a constraint in the queue is done by finding a position preserving the ordering $\mathcal{O}$ starting from the head. The field `Already_In_Queue` prevents us from pushing a constraint in the queue more than once.

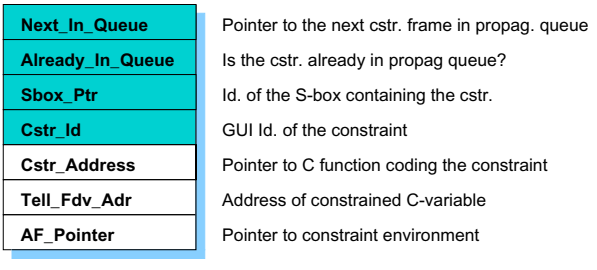


Fig. 11.13. Modified constraint frame

The register SBC contains the name of the current S-box. This permits to know easily to which S-box a newly created constraint belongs to: at the creation of a constraint frame, the contents of SBC is copied into the field `Sbox_Ptr`.

The last field, `Cstr_Id`, is used as an identifier for the constraint when communicating with the [incr Tcl] interface. It permits, e.g., to ask the interface to highlight the graphical representation of the constraint each time it is reinvoked by the solver.

The original variable frame has been but slightly modified since the only modifications lie in the removal of the unused fields, namely all the ones which where used to link the frames into a propagation queue. The resulting frame is depicted in Figure 11.14.

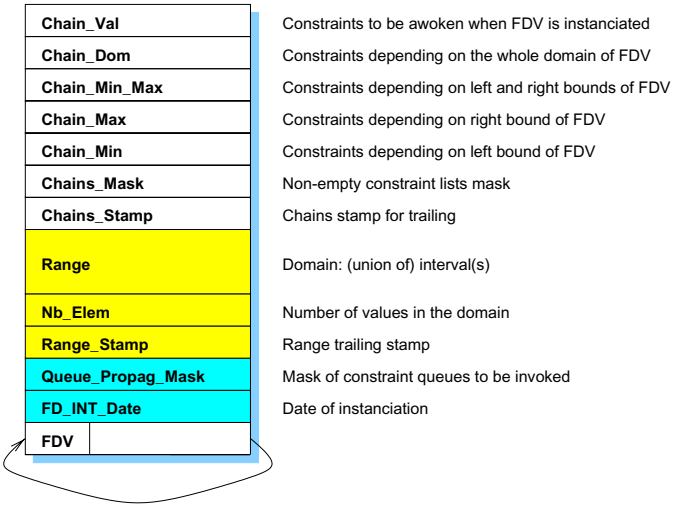


Fig. 11.14. Modified variable Frame

11.6.3 Handling the S-Boxes

S-boxes are created by tagging with the mouse some goals in the debugger editor (see Section 11.5). Before being interpreted, the code contained in the editor is merged with those tags in a new CLP program in the following way: for each tag on a set of goals/constraints $g_1, \ldots, g_m$, the instructions `begin_sbox`[2] and `end_sbox` are added, respectively, before g_1 and after g_m. They are in charge of creating and "closing" at runtime the corresponding S-box.

Example 11.6.1. Translation of a tagged program

```
p:- a, b, c, d.  |  p:- a, begin_sbox, b, c, end_sbox, d.
```

Each S-box is uniquely identified by its Dewey index (see Section 11.4.2). A natural representation of this identifier is a character string. Thus, the manipulated "S-box names" have, for example, the form `"1.2.3"`. An advantage of such a representation lies in that the implementation of the ordering $\mathcal{O}$ becomes a straightforward comparison of strings.

The identifier of a S-box to be created is determined as follows: a "last sons" stack (see Figure 11.15) contains the number of sons of each S-box along with some additional information used for backtracking purposes; the name of a new S-box is then the concatenation of the contents of the field `nameSB`

[2] The instruction `begin_sbox` accepts an argument which is the symbolic name of the S-box chosen by the user. It is omitted here for the sake of clarity.

on top of the stack (the current S-box) with the number of sons incremented by one.

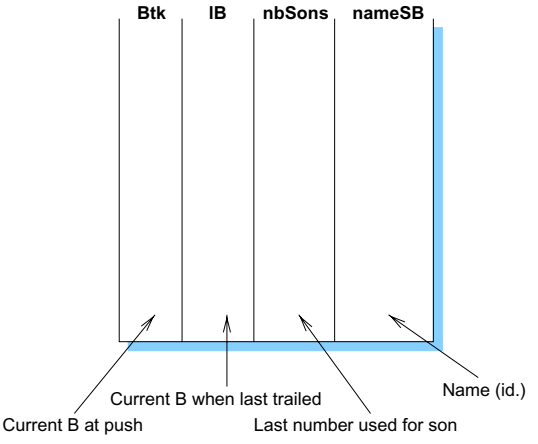


Fig. 11.15. "Last sons" stack

On creation of a new S-box σ, the focus is changed to σ since the propagation cannot be done outside of σ until it is closed (otherwise, σ could not be viewed as a constraint like the primitive ones). It is then necessary to keep a stack of the names of the S-boxes having the focus in order to be able to reset the focus to its previous value when a S-box is closed. This is done by the SBF *stack* described in Figure 11.16.

Note that the S-box identifiers are not duplicated in the different structures (SBF stack, last sons stack, ...) since they hold only pointers to them. As a consequence, it is easy to free the memory space used to represent these strings on backtracking, using the field `sbList` of the S-box trail.

11.7 Conclusion

We have presented the S-box abstraction that permits introducing in the store the semantics of sets of constraints that is present in the program. S-boxes allow us to design a new kind of debugger that focuses on the constraint store and the propagation process of variables' domains narrowing. Using S-boxes, the store is no longer a huge and flat collection of constraints but a tree structure with few "user's constraints". The programmer may then inspect some particular set of constraints by zooming in and out the S-boxes containing them. The inspection of the store is made easier by the fact that these global constraints have the semantics the programmer gave in first place to the corresponding sets of constraints in its source code.

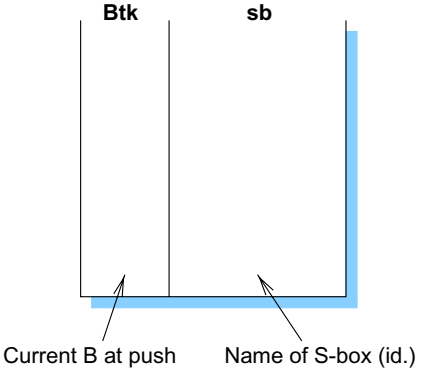


Fig. 11.16. SBF stack

S-boxes address both Correctness Debugging and Performance Debugging issues: gathering some constraints into an S-box allows the user to verify the conformance of them with what he had in mind by simply comparing the input variable domains with the output ones; on the other hand, using S-boxes permits collecting statistics for some sets of constraints (e.g. the number of domain reductions their re-invocations performed).

Since S-boxes rely only on the "constraint narrowing operator" notion—that has been shown to be able to model most of the constraint solvers—and on the NAR algorithm, they may be used without modification on linear as well as non-linear constraint solvers, and for integer-valued as well as real-valued variables.

The S-box abstraction required the modification of the propagation algorithm used in constraint solvers. We have given an incremental version of such an algorithm which is only a slight modification of the traditional algorithm, thus permitting to replace it smoothly by our version supporting S-boxes. A prototype of a debugger with S-boxes has been implemented in order to validate our ideas. It appeared that the required modifications of the host solver were few, which is encouraging for future portability on other solvers.

As said previously, inspecting the store is not sufficient, and debugging must be done at the Prolog level as well. At present, our prototype supports Prolog debugging by simply using the text-based Prolog debugger of clp(fd). However, we believe that these two tools are not as connected as they should be. In particular, our system lacks of a powerful graphical Prolog debugger where data in the source code (constraints, variables) would be connected to their representation in the store. This is a planned enhancement.

The method used to support S-boxes is mainly based on modifying the propagation order for re-invocation of constraints. It is then tempting to

relate it with those presented by Borning *et al.* [11.4] and Wallace and Freuder [11.24]:

- Borning suggests separating constraints into classes according to some priority criterion. For example, one can assign a high priority to mandatory constraints, and a low one to constraints that are mere preferences. Such a classification is achieved by labelling constraints with their priority level. This labelling process could be modelled by S-boxes; one would only have to redesign the propagation algorithm in order that constraints with the highest priority—collected in the same S-box—be reinvoked first;
- On the other hand, Wallace and Freuder [11.24] have shown that considering in a precise order constraints to be reinvoked could speed-up the quiescence of the constraint network, and then the solving process. For example, this is the case when first considering constraints where occur variables with many occurrences in the store. Such a strategy could be modelled by S-boxes. Extending the S-box abstraction to permit the use of optimising heuristics is a direction for future researches.

Acknowledgements

The research exposed here was supported by the LTR DiSCiPl European ESPRIT Project #22532. Discussions with the researchers involved in this project are gracefully acknowledged.

References

11.1 A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In: F. Benhamou and A. Colmerauer (eds.): *Constraint Logic Programming: Selected Research*. The MIT Press, pp. 421–435, 1993.

11.2 F. Benhamou. Heterogeneous Constraint Solving. In: *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*. Aachen, Germany, pp. 62–76, 1996.

11.3 F. Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In: *JFPL'95: IV^e Journées Francophones de Programmation en Logique*. Dijon, France, pp. 51–65, 1995.

11.4 A. Borning, B. Freeman-Benson, and M. Wilson. Constraint Hierarchies. *LISP and symbolic computation* **5**, 223–270. Kluwer Academic Publishers, 1992.

11.5 J. Burg, C. Hughes, J. Moshell, and S. Lang. Constraint-based Programming: A Survey. Technical report IST-TR-90-16, Dept. of Computer Science, University of Central Florida, 1990.

11.6 P. Codognet and D. Diaz. Compiling Constraints in clp(fd). *Journal of Logic Programming* **27**(3), 185–226, 1996.

11.7 A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

11.8 J. Darlington and Y.-K. Guo. A New Perspective on Integrating Functions and Logic Languages. In: *3rd International Conference on Fifth Generation Computer Systems*. Tokyo, Japan, pp. 682–693, 1992.
11.9 D. Diaz, D. Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système `CLP(FD)`. Ph.D. thesis, Université d'Orléans, 1995.
11.10 R. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence* **1**, 27–120, 1970.
11.11 F. Goualard, F. Benhamou, and L. Granvilliers. An Extension of the WAM for Hybrid Interval Solvers. *Journal of Functional and Logic Programming, 4. The MIT Press*. Special issue on Parallelism and Implementation Technology for Constraint Logic Programming, V. Santos Costa, E. Pontelli and G. Gupta (eds), April, 1999.
11.12 H. Hong. Non-linear Real Constraints in Constraint Logic Programming. In: H. Kirchner and G. Levi (eds.): *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, Proceedings*, Vol. 632 of *Lecture Notes in Computer Science*. Berlin-Heidelberg-New York, pp. 201–212, September 1992.
11.13 J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming* **19**(20), 1994.
11.14 D. E. Knuth. *Fundamental Algorithms*, Vol. 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.
11.15 A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence* **1**(8), 99–118, 1997.
11.16 M. J. McLennan. The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More. In: USENIX Association (ed.): *Proceedings of the Tcl/Tk Workshop, July 6–8, 1995, Toronto, Ontario, Canada*. Berkeley, CA, USA, pp. 151–160.
11.17 M. Meier. Debugging constraint programs. In: U. Montanari and F. Rossi (eds.): *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, Vol. 976, pp. 204–221, 1995.
11.18 R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence* **28**, 225–233, 1986.
11.19 W. J. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In: F. Benhamou and A. Colmerauer (eds.): *Constraint Logic Programming: Selected Papers*. The MIT Press, 1993.
11.20 J. K. Ousterhout. *Tcl and the Tk Toolkit*, Professional Computing Series. Addison-Wesley, 1994.
11.21 J.-F. Puget. A C++ implementation of CLP. In: *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94)*. Singapore, 1994.
11.22 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In: L. Naish (ed.): *Proceedings of the 14th International Conference on Logic Programming*. Cambridge, pp. 286–300, 1997.
11.23 D. Singmaster. Quelques divertissements numériques. *La Recherche* **26**(278), 818–823. Numéro Spécial Nombres, 1995.
11.24 R. J. Wallace and E. C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In: *Proceedings of CAI'92*, 1992.

12. Complex Constraint Abstraction: Global Constraint Visualisation

Helmut Simonis, Abder Aggoun, Nicolas Beldiceanu, and Eric Bourreau

COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex, France
email: Helmut.Simonis@cosytec.com

In this chapter we describe visualisation tools for global constraints in the CHIP system. These tools help to understand the behaviour of different global constraints, the core feature of the CHIP constraint language. The tools follow a general classification scheme for the use of global constraints in global constraint concepts. Each concept captures the use of a constraint for a particular type of problem, which can be visualised in a specific way. The different visualisers form a class structure of CHIP++ objects, and can be extended or modified by rewriting some callback predicates.

12.1 Introduction

Debugging of constraint programs take can quite different forms and requires a wide variety of tools. Depending on the host language, the programmer may use a standard debugger or use trace facilities built into the language. It has been reported that debugging with such tools is often time consuming and not very effective [12.4]. To overcome this problem, ad hoc graphical output has been generated already for the very first constraint programs. In recent years, graphical tools which simplify this task and which make the debugging process more systematic have been developed, in particular for Eclipse [12.6], OZ [12.7] and CHIP [12.9]. We can distinguish the following variants:

- search-tree tools (see chapters 6,7,8) represent the search-tree generated in a finite domain program in a graphical form and allow user interaction to query the state of the engine at each point. The OZ explorer also provides interactive search facilities, the user can expand nodes in the search tree under his control and thus solve a problem "by hand" while the system performs the constraint propagation. The CHIP search-tree tool (see chapter 7) generates the search-tree in a first phase, and allows interaction by re-creating the state of a particular node by re-instantiating variables to certain values.
- The domains of domain variables can often be graphically displayed. Most often this display takes the form of an incidence matrix variables-values, but some systems also use a textual representation.
- Display of the propagation steps in the constraint solver is less common. In the CHIP search-tree tool, each propagation step after a value assignment

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 299–317, 2000.

is shown in a graphical form, which indicates the variables affected and the type of update.
- Failure analysis, an important aspect of understanding constraint search behaviour, can also be supported by graphical tools.
- The general form of the constraint network is often interesting. Little work has been reported on the display of a constraint network in graph form. The CHIP search-tree tool uses an incidence matrix representation, which is particular useful for the representation of non-binary, global constraints.

In this chapter we describe another approach to visualisation, the concept based representation of global constraints. Global constraints form the core of the CHIP system, they are high-level abstractions which can be used to express complex conditions between sets of variables in a simple way. They also contain powerful propagation methods to reduce domains and to check consistency. For the development of the visualisation tools, the following design aims were stated:

- Multiple use: Each tool must be applicable in multiple instances, so that it can be re-used for many different application problems.
- Use in the search-tree tool: The tools must work together with the search-tree tool. If a node in the search-tree tool is selected, the constraint visualiser should display the state of the constraint in this node.
- Use as debugging tool without the search-tree tool: At the same time, the tools will be used outside the search-tree tool. An API (application programmer's interface) is provided so that users can update the display at particular points in the application.
- Use as part of application framework: The tools must integrate into the application framework developed by COSYTEC [12.5]. For simple demos, they can be embedded into applications as sub windows which are defined via the panel package. For complex applications, they can be added as development phase extensions of the application framework.
- Explanation facilities: The tools should try to explain which constraint propagation method has caused a change in the display, so that users can understand problems in models more efficiently.
- Visualise concepts, not code: The global constraints are powerful abstractions, but they do not always correspond to the programmer's view of a particular problem. The same constraint can be used to express many different application concepts. The visualisation tools should try to show the constraints in the form of these concepts. This means that one global constraint may have more than one visualisation.
- User extendible by rewriting some callbacks: The tools can easily be extended by re-writing/extending some basic callbacks. Rather generic tools can be adapted more closely to particular application domains without major rewrites.

12.2 Global Constraint Concepts

In this section we briefly review the difference between global constraints and constraint concepts. Global constraints [12.1][12.2] are abstractions which look for a compromise between expressive power and structure. To increase expressive power, the constraint should be as general and flexible as possible, while the structure is needed to allow efficient constraint propagation. In practice, global constraints are usually as general as the methods used inside allow. While this generality increases the usefulness of the constraint and limits the number of different constraint types in a system, it also increases the difficulty of learning and effectively using the constraint. For many applications, it is not necessary for example to understand all 25 pages of the `cycle` documentation to use the constraint. The constraint concepts see the problem from another perspective. Each typical use of a constraint forms a constraint concept, these concepts can be combined in an application in different ways to solve a problem. Some concepts are simply different interpretations of the same constraint, some others are restrictions of a constraint, for example limiting degrees of freedom in some parameter. We will now describe the constraint concepts used in the CHIP environment.

12.2.1 Cumulative

Some of the concepts for `cumulative` were already discussed in the original paper introducing the constraint [12.1]. At the moment, we use the following concepts:

– cumulative resource,
– disjunctive resource,
– bin packing,
– producer consumer,
– redundant projection

The cumulative resource is the most commonly used concept for `cumulative`. The items in the constraints are seen as tasks with `start` time, `duration` and `resource` consumption. The x-axis is the time axis, the y-axis is the resource usage. Limits can be placed on the overall end and/or the maximal resource utilisation. The disjunctive resource case is a special case of the cumulative resource with a resource limit of 1, where each task has a resource usage of 1. The bin packing concept for cumulative sees tasks as items to be packed into bins. The x-axis denotes the different bins, the y-axis the height of the bins. Each item is specified by its bin assignment, a width which is equal to 1 and a height which corresponds to the size of the item. Producer/Consumer constraints were introduced in [12.8] to describe the behaviour of consumable resources. Tasks either start from time 0 and block a resource amount up to a time point on the x-axis (producers) or they start from a time point and

stretch to the end of the time period (consumers) and consume a given amount of resource. The redundant projection is used to strengthen a constraint model by projecting rectangles of a (2D) placement problem onto one axis.

12.2.2 Diffn

For the `diffn` [12.2] constraint, a large number of concepts is known.

- disjunctive tasks,
- machine scheduling,
- assignment,
- placement 2d,
- placement with spare,
- placement 3D,
- placement 3D + assignment

The `diffn` constraint can be used to express disjunctive scheduling problems or multiple machine scheduling problems. The x-dimension denotes time, the y-dimension machine allocation. All tasks have `height` 1 to indicate that they use one machine during their execution. Related is the assignment problem, where tasks fixed in time must be allocated to different machines. The `diffn` constraint can also be used for placement problems. The easiest variant is a two-dimensional placement. More complex are the cases where we can use a spare dimension for relaxation, or where we place objects in a three-dimensional space. The last concept uses four dimensions to describe packing problems with multiple containers. One dimension is used to assign an item to one container, the remaining three dimensions control the placement of the item in that container.

12.2.3 Cycle

The `cycle` constraint [12.2] also is used in many different ways:

- oriented graph,
- non oriented graph,
- geographical tours,
- tours with machine assignment,
- tours with fixed times,
- tours with time windows,
- loading/unloading
- scheduling with product dependent set-up times,

In its basic form, it is used to find cycles in oriented or non-oriented graphs. If the locations and distances are derived from geographical data, we want to build geographical tours. Additional parameters allow the introduction of machine assignment to the problem. In other problems, start times

are linked to the nodes and we are interested in tour planning with time windows. A significant special case is the tour planning with fixed time windows, for example in the case of railway/aircraft rotations. Another extension handles capacity of the tours together with loading/unloading operations at each node. A completely different model uses the `cycle` constraint to express set-up times in multi-machine scheduling problems.

12.2.4 Among

We also use the `among` constraint [12.2] in a number of different ways:

– single among,
– overlapping sequences,
– extending sequences,
– multiple among

In the basic constraint, the number of occurrences of a set of values in a list of variables is limited. This constraint can also be applied to overlapping subsequences or to increasing sequences from a start. In its final form, multiple among constraints on different value sets are handled together in the multiple among concept.

12.3 Principles of Operation

In this section we describe the basic implementation structure of the visualiser tools. If you are not interested in extending/modifying some visualiser tool, you may want to skip this section.

12.3.1 Class Structure

The class visualise is obtained by adding the `list_entry` attributes to the `visualize1` class. All abstract classes are shown *italic* in figure 12.1 below, they should never be used by the application programmer. You can add new classes either by deriving a new class from an abstract class, or by specialising an existing class.

The classes `visualize_ltsrsb` and `visualize_srsb` differ mainly in their window layout. The first (see figure 12.2) has a main drawing area, drawing areas to the left and on top and scrollbars to the right and at the bottom. Most structured displays are derived from this class, the left and top area being used for scales or resource display.

The `visualize_srsb` class (see figure 12.3) has only one main drawing area which is affected by the scrollbars to the right and at the bottom. It is mainly used for drawing concepts of a simpler form.

visualize

- *visualize_ltsrsb*
  - *visualize_gantt*
    - visualize_disjunctive_tasks,
    - visualize_machine_scheduling,
    - visualize_assignment,
    - visualize_placement_2d,
    - visualize_placement_remains
  - *visualize_profile*
    - visualize_cumulative_resource,
    - visualize_disjunctive_resource,
    - visualize_redundant_projection,
    - visualize_bin_packing,
    - visualize_producer_consumer
  - *visualize_domain*
    - visualize_variable
  - *visualize_measure*
    - *visualize_stack*
      - visualize_local_stack
      - visualize_global_stack
    - visualize_time
- *visualize_srsb*
  - *visualize_drawing*
    - *visualize_graph*
      - visualize_oriented_graph,
      - visualize_non_oriented_graph,
      - visualize_geographical_tours
      - visualize_graph_lines

Fig. 12.1. Visualize class structure

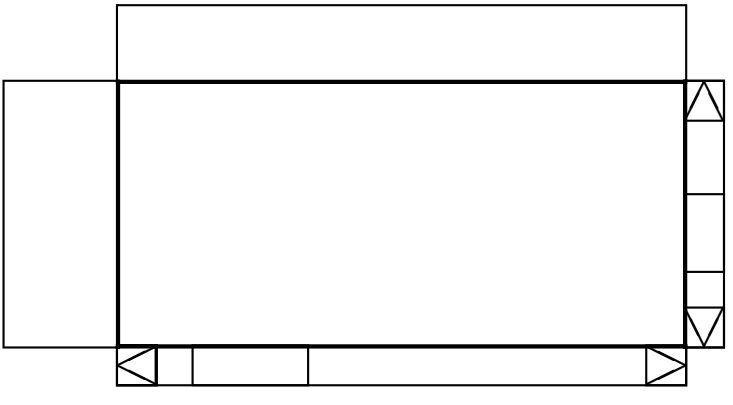


Fig. 12.2. Visualize_ltsrsb window layout

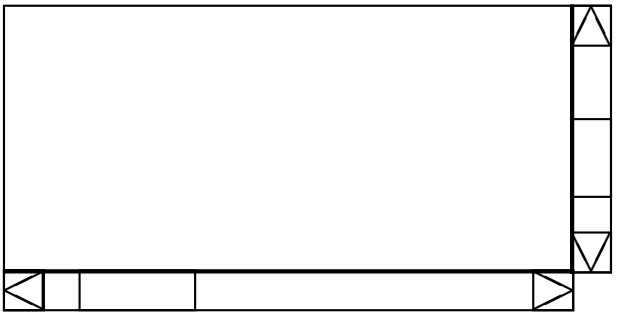


Fig. 12.3. Visualize_srsb window layout

12.3.2 Callbacks

The callbacks are stored in attributes of the objects in the form of predicate calls `pred(X)`, where `pred/1` is the callback name and X is the name of the object affected. Overwriting such a callback with a new routine means that the new code is executed instead of the old one. This requires that the new code should emulate all actions of the old code that you want to keep. Alternatively, the code can be structured in such a way that the new code calls the callback of the parent class before/after executing any of its own routines.

create_windows. This code creates the object, sets attributes and creates all windows of the visualiser. It does not draw in the new windows.

resize_callback. This code is executed if the back window of the visualiser is resized. This is used to keep the scrollbars, title and resource windows in constant size after a resize.

reset_callback. This code is executed when an existing visualiser is re-created during program execution. It should make sure that default values are set correctly.

init_callback (user code). The init callback is responsible to calculate the displayed area for the constraint. This area should be stored in the `x1, y1, x2, y1, ix1, iy1, ix2, iy2` attributes. The code should not attempt to change the vdc of the windows, that is done in another part of the program. Most visualisers want to modify this code.

draw_callback (user code). The callback is responsible to draw the visualisation. The drawing can use all attributes of the visualiser, but most important will be the items attribute which contains the current set of items. These values must not be modified by code.

resource_draw_callback (ltsrsb only). This code draws the resource scale. The default is a numeric scale dependent on the vdc space of the visualizer. This callback only exists for ltsrsb subclasses.

title_draw_callback (ltsrsb only). This code draws the title scale. The default is a numeric scale dependent on the vdc space of the visualiser. This callback only exists for ltsrsb subclasses.

12.3.3 API

This section describes how to use the visualiser library as an application programmer.

Creation. A visualiser object is created by a wrapper around a constraint call inside a CHIP program. The wrapper has the general form

```
visualize(Constraint, Visualizer_type, Attribte_list, Name)
```

The first argument is the call to the constraint, the second argument is the type of visualiser used (an atom), the third is a list of attribute value pairs of the form `A<-B` where A is an attribute of the visualiser and B is the value used. The last argument is the name of the visualiser object. If the constraint call is executed only once in the application, this name should be an atom. If the constraint call is executed multiple times in a recursive loop, then a new name must be generated each time. A typical example is the call

```
visualize(cumulative(Start, Dur, Res, unused, unused,
            Limit, End, unused),
          visualize_cumulative_resource,
          [winw<-500, winh<-400],
          cumulative1)
```

which generates the `cumulative_resource` visualiser `cumulative1` working on a `cumulative` constraint and setting values for the window width and height in the tool.

When the constraint is called, the following actions are performed:

- The program generates the visualiser object of the correct class.
- It then creates all windows required.
- All the parameters are set either to their default value or (if given) to the value specified in the attribute list.
- The constraint is posted. If the constraint fails immediately, the state of the constraint at this time is shown. If the constraint delays, nothing is drawn. This particular mechanism is used to allow failure analysis at set-up time.
- The internal constraint number is remembered. This number is created internally in the constraint solver and is increased every time a new constraint is posted. Remembering the value allows us to refer to a constraint from the search-tree tool.
- The constraint call is remembered by storing the call as an variable attribute in the object. Whenever the domain variables in the call change, the new information can be extracted and displayed. If the program ever backtracks over the creation of the constraint visualiser, no further updates of the display are possible.

Note that the objects are not removed when the query terminates. If the same program is run again, the system recognises the names of existing visualisers and attaches the constraints to the existing windows. In this way, settings on window positions and sizes are not lost from one query to the next. During a debugging session, the same code can be executed multiple times without destroying and recreating the visualisation tools.

Update. To update the tools, two predicates have been defined. Note that the visualiser tools are also updated automatically inside the search tree tool. In that case, none of these predicates need to be called in the application program.

```
visualize_update
```

This predicate updates all visualiser tools that are active at this moment. This is the normal way to update the display at a given point in the search.

```
visualize_draw(X)
```

This predicate updates only the visualiser with the name X. This predicate is useful to display the state of some constraint at different points during the search process, for example to compare the effect of one assignment step.

12.4 Interface to Search-Tree Tool

The CHIP search-tree tool is described in chapter 7. Its function is to capture and visualise the form of a search-tree generated by a CHIP search routine. In the most simple case, a built-in search procedure (labelling) is replaced by another built-in, which automatically calls the correct primitives. In a more complex situation, a user written search routine must be annotated with some special predicates to identify how the search should be displayed. The interface from the search-tree tool to the global constraint visualisers is transparent. If a visualisation tool is active during the search, it will be updated whenever a new node in the search-tree is selected. The state of the constraint will be displayed for the node which has been selected.

We now describe what is happening in detail when a node is selected. When the callback on the selection of a node is executed, the following steps are performed:

- The state of the search in this node is re-instated. For this, the path from the root to the node is scanned to find the variable bindings that must be restored. By constraint propagation, the state of all domain variables and constraints is updated.
- The node specific information is displayed in the search-tree tool. Depending on the selected view, this may be the display of all variables or of all constraints, etc.

– All currently enabled visualisers are redrawn in the current state of the search.
– The bindings of all variables are undone, we fail until the root of the tree.

12.5 Use Outside Search-Tree Tool

The global constraint visualisers can be used outside the search tree tool. In that case, the programmer is responsible of calling the update routine (`visualize_update/0`) at the right point in the search process. If called at every point in tree search, i.e. after each `indomain`, the complete evolution of the constraint is shown. This is useful to obtain a first impression of the behaviour of the constraint, but a more detailed analysis will be required to understand failures or missing propagation at some particular step in the search.

12.6 Cumulative Visualisers

We present different visualiser tools in detail. For space reasons, we can not discuss all visualisers that have been implemented, we restrict ourselves to typical examples.

12.6.1 Cumulative Resource

Probably the most often used visualiser represents cumulative resources. In this tool, we show three types of objects, profiles, limits and tasks. Figure 12.4 shows the cumulative resource visualiser for a machine in a scheduling example from the Alvarez benchmark set [12.3]. We will now explain the different components shown in the diagram.

Profiles. There are three different profiles that can be generated from the tasks in the constraints. Each of these profiles approximates the final resource profile, that is generated when all tasks are fixed. The fixed profile (black) is generated from all tasks which are already assigned, i.e. for which start, duration and resource use are integer values. The profile adds up the resource use for each time point. It is given by the function

$$y(t) = \sum_{i=1 \mid start_i \leqslant t < start_i + duration_i}^{n} resource_i$$

where t ranges over the interval

$$[\min_{i=1}^{n} start_i, \max_{i=1}^{n} start_i + duration_i]$$

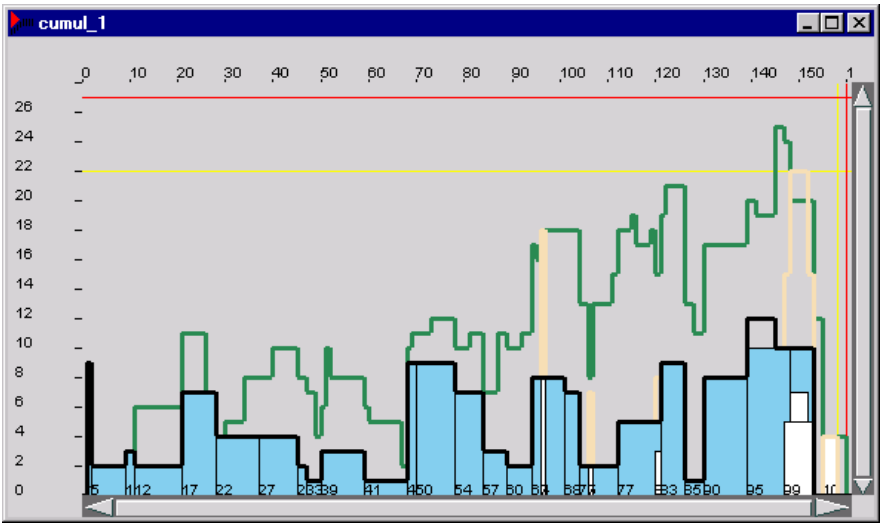


Fig. 12.4. Cumulative resource view

The obligatory part profile (white) is generated from all tasks for which an *obligatory part* is known. This is the case if the latest start and the earliest end of a task form a non-empty interval. This means that the task is known to be active in this time interval, even if neither start nor duration is fixed. The obligatory part profile is always above the fixed profile, and always below the overall resource limit. This is one of the necessary conditions that the cumulative constraint checks for satisfiability. The profile is given by the formula

$$y(t) = \sum_{i=1 \mid start_i^{max} \leqslant t < start_i^{min} + duration_i^{min}}^{n} resource_i^{min}$$

The notation $start_i^{max}$ denotes the maximum value in the domain of the *start* variable of task i.

If the domains for the start variables are very big, no obligatory parts will exist, but if by constraint propagation the domains are restricted, the obligatory parts start to appear and are taken into account in the constraint propagation. When all tasks are fixed, the fixed task profile and the obligatory part profile are identical and describe the total resource use for all tasks.

We display another profile in the cumulative resource visualiser, the *expected resource use* profile (gray). For this profile we distribute the resource use of a task over the total time period when it can be placed, from the earliest start to the latest end. This approximation of the resource use gives us some indication of resource requirements, even if no obligatory parts exist. The profile is given by the formula

$$y(t) = \sum_{\substack{i=1| \\ start_i^{min} \leqslant t \\ t < start_i^{max} + duration_i^{max}}}^{n} \frac{resource_i^{min} * duration_i^{min}}{duration_i^{max} + start_i^{max} - start_i^{min}}$$

The expected profile does not always stay below the overall resource limit and the final profile may be below the expected profile in some intervals. But the profile is still useful, as it provides important negative information. If the expected profile is zero for some time period, we know that no task can be scheduled in this interval.

Limits. For both the overall end and the overall resource limit variable we display the minimum and the maximum values as red and yellow (vertical and horizontal) lines. These limits are also used to determine the overall drawing area for the constraint.

Tasks. To indicate the space taken up by each task, they are displayed as gray rectangles as soon as they are completely fixed. If an obligatory part exists, it is displayed as a white rectangle. All tasks are drawn from the bottom of the display, possibly obscuring other tasks. It is unfortunately not possible in the general case to place them as rectangles which do not overlap and which fit under the resource profile.

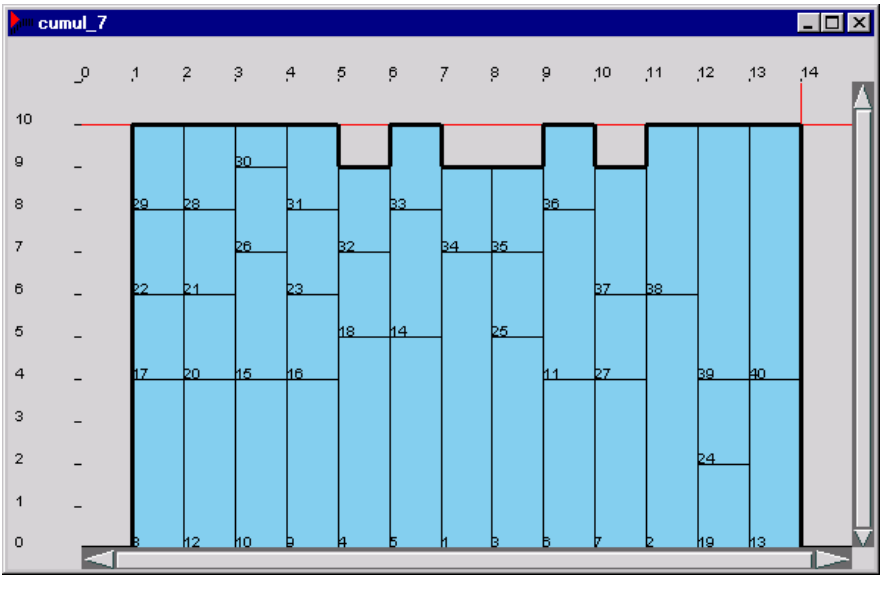


Fig. 12.5. Bin packing example

12.6.2 Bin Packing

The bin packing visualiser is derived from the cumulative resource visualiser. In this particular case, all task duration values are equal to one. The fixed part profile is generated in the same way as for the cumulative resource tool. As all the task duration are one, the concept of obligatory parts does not exist. Instead, we can place all tasks non-overlapping under the resource profile. For this, we define an additional domain variable for each task in the visualiser, its y-coordinate in the visualiser display. The x-coordinate is given by the `start` variable in the `cumulative`, the width is one and the height is equal to the `resource` value in the `cumulative`. We then state a `diffn` constraint to place the task rectangles without overlapping. Each time we update this visualiser, we again solve the small constraint problem placing the tasks. Figure 12.5 shows an example bin packing visualiser. All tasks have been placed and are drawn underneath the generated profile. It is interesting to note that we use the constraint system itself to visualise some constraint. We use this feature for other visualisers as well.

12.7 Diffn Visualisers

The `diffn` constraint has a number of different uses, but the most common is the 2D placement concept. An interesting dual visualiser is the `placement_remains` tool, which shows all tasks which are not yet drawn in the placement visualiser.

12.7.1 Placement 2D

The 2D placement visualisers shows a set of 2D rectangles in an area which is defined by the minimal and maximal values in the domains of the rectangles. If a rectangle is fixed, i.e. x, y, $width$ and $height$ values are integers, then the rectangle is drawn in gray. If an obligatory part exists, defined in analogous way to the obligatory parts in `cumulative`, then the obligatory part is drawn in white. If an object is selected in another visualiser, the area in which it can be displayed in the `diffn` will be highlighted during the selection. We only use the bounds on the coordinates for this purpose, eventual holes in the domain are not shown. Figure 12.6 shows a typical visualiser example. Three rectangles are already placed, and there are two obligatory parts shown.

12.7.2 Placement Remains

The 2D placement tool is very useful if many tasks already have been placed. But in the very beginning no tasks will be placed. How can we get an idea about the rectangles that have to be placed? The `placement_remains` tool

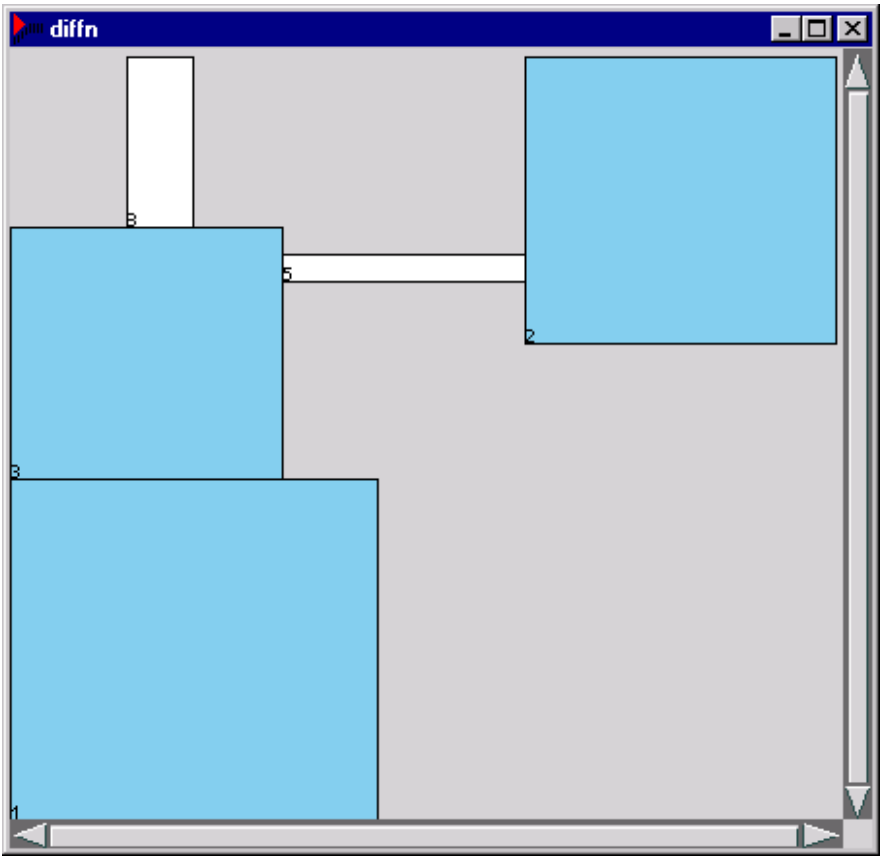


Fig. 12.6. 2D placement

provides this information. It displays all rectangles which are not completely placed. Rectangles for which an obligatory part exists are drawn in white, all others are drawn in gray. When we select one of the rectangles, the area in which it can be placed in the placement problem will be highlighted in the `placement_2D` visualiser.

The layout of the rectangles in this tool again is controlled by a constraint program, which itself uses a `diffn` constraint to place all remaining, unassigned rectangles. We do not enforce the domain limits of the rectangles at this point and only ensure that the rectangles are not overlapping. Figure 12.7 shows a snapshot of the `placement_remains` tool.

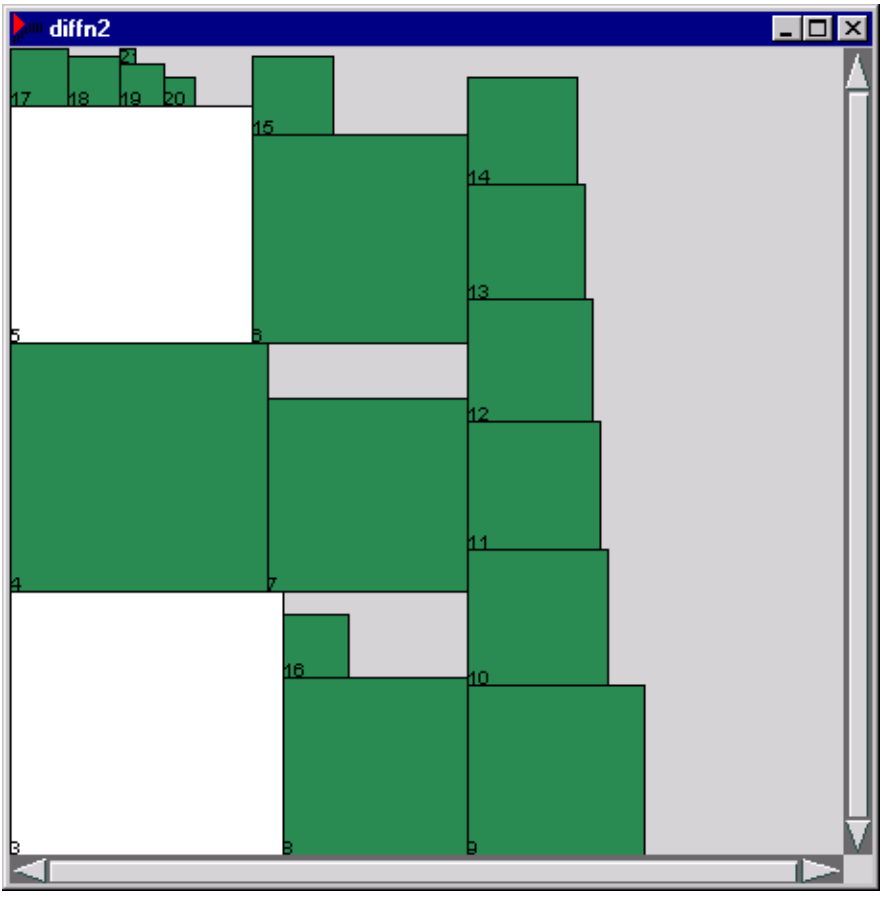


Fig. 12.7. Showing unplaced items

12.8 Cycle Visualiser

We show now two quite different interpretations of the `cycle` constraint, one based on a geographical representation, the other on the idea of lines in the directed graph defined by the constraint.

12.8.1 Geographical Tour

In this tool, the directed graph given in the `cycle` constraint is shown by placing the nodes at co-ordinates which are given by the user as an additional attribute. Typically, these locations will correspond to some geographical location. The connections between nodes are drawn as arrows. If there is only one successor, the arrow is drawn in light gray. If there are still multiple possible successors, the arrows are drawn in dark gray. An additional attribute,

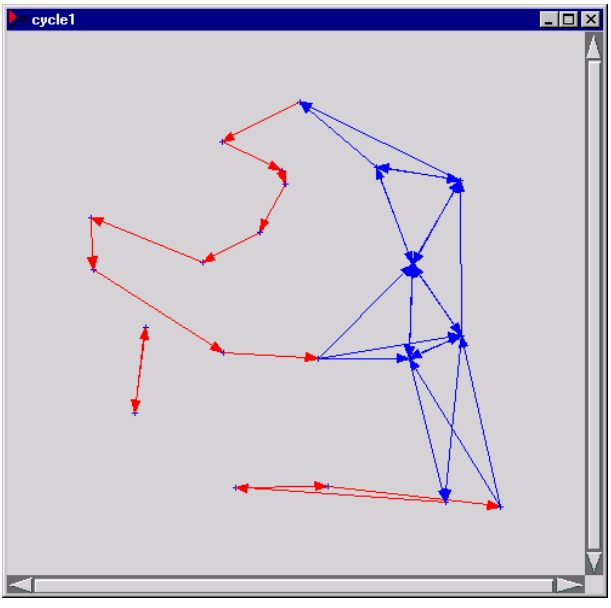


Fig. 12.8. Geographical tour visualiser

`show_domain`, controls the display. If the domain of a node is larger than this value, no successors will be drawn. This helps to identify small, restricted parts of the graph without drawing large numbers of connections to clutter the display. Figure 12.8 shows an example. A number of nodes are already assigned, there is one closed cycle in the bottom left corner, but the domains of other nodes consist of rather small sets of nodes. By constraint propagation, all links between nodes that are far apart have already been removed.

12.8.2 Graph Lines

The next visualiser uses a different representation of the `cycle` constraint, which is based on the idea of a graph line. A line starts with a node, which has not yet been chosen as the successor for another node and ends with a node for which the successor is not yet given. Initially, all nodes form lines of their own. Whenever a variable is assigned, two lines are merged or a line is closed and a cycle is created. When all variables are assigned, a given number of cycles will have been generated. Figure 12.9 shows an example, which displays seven lines and one completed cycle. The cycle (drawn in gray) consists of two nodes, numbered 9 and 20. Five lines consist only of one node each. The last node in the line is drawn in white and outlined in black, with the domain of the successor variable printed at the end of the line. In the current state, no line can be closed, as for each line the start node is not in the domain of the end node. The next assignment step will therefore consist in the merging

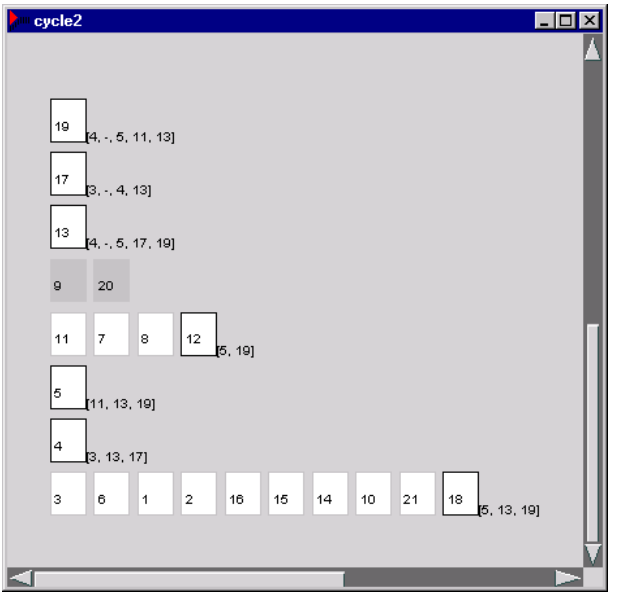


Fig. 12.9. Graph lines visualiser

of two lines. At each display, the layout of the lines is re-calculated. This visualiser can quickly give a good idea how the assignment routine is working and whether constraint propagation is forcing the merging of lines.

12.9 Interaction between Visualisers

When considering multiple visualisers at the same time, it is useful to find the relation between the different items displayed. Figure 12.10 shows an example. We have a two `redundant_projection` visualisers (similar to the `cumulative_resource` tool described above) on the left, one `diffn placement_2d` tool one the top right and one `diffn placement_remains` tool at the bottom right. Selecting a rectangle in the `placement_remains` tool highlights its position in the other tools. At the moment, the link is done only by argument position. This interaction will be extended to allow customised links between entries in different visualisers.

12.10 Conclusions

In this chapter we have described different visualisers for global constraint concepts. One global constraint, like `cumulative`, can be used in multiple

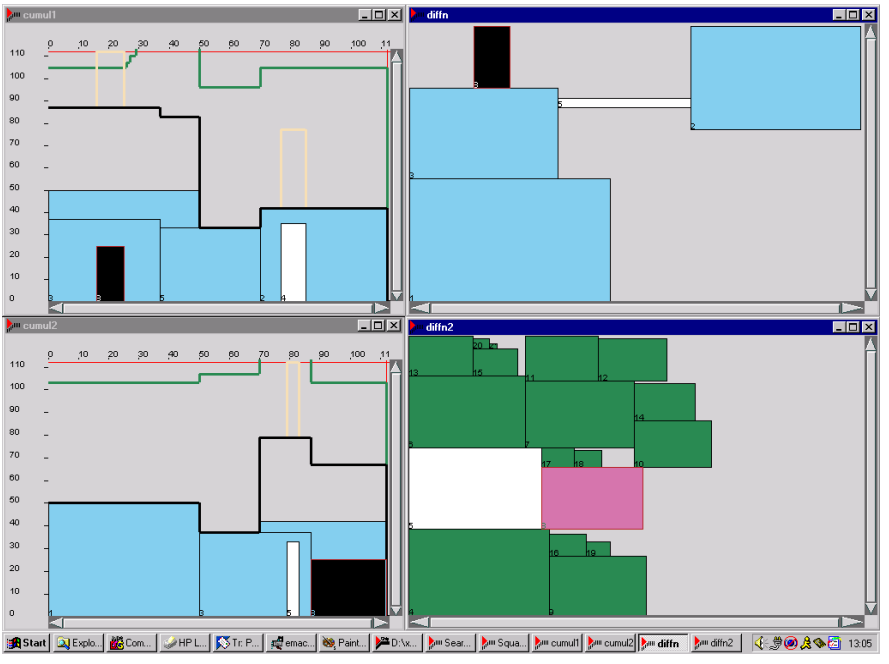


Fig. 12.10. Visualiser Interaction

contexts, to solve different types of constraint concepts. To understand the behaviour of the constraint, it is best to describe the propagation in terms of these concepts as well. The global constraint visualisers are all derived from a fundamental class of visualiser objects. The user can overwrite different attributes and methods to adapt the tool to his particular needs. Working together with the search-tree tool or as a stand-alone utility, these visualisers allow a better understanding of the global constraints and an easier exploitation of their functionality.

Acknowledgement

The work presented here is part of COSYTEC's work package in the DiSCiPl project and was developed using ideas from a number of consortium partners, in particular from UPM. Feedback from a number of early users at COSYTEC was also very valuable.

References

12.1 A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems. Journal of Mathematical and Computer Modelling,

Vol. 17, No. 7, pages 57-73,Pergamon Press, 1993.
12.2 N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.
12.3 N. Beldiceanu, E. Bourreau, D. Rivreau, and H. Simonis. Solving Resource-Constrained Project Scheduling Problems with CHIP. Fifth International Workshop on Project Management and Scheduling, Poznan, Poland, April 1996.
12.4 M. Fabris et al. CP Debugging Needs and Tools. In Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97, Pages 103-122, Linköping, Sweden, May 1997.
12.5 P. Kay and H. Simonis. Building Industrial CHIP Applications from Reusable Software Components. In Proceedings of the Third International Conference on the Practical Application of Prolog, Paris, France, April 1995.
12.6 M. Meier. Debugging Constraint Programs. In Principles and Practice of Constraint Programming, page 204-221, Cassis, France, September 1995, Springer-Verlag, Lecture Notes in Computer Science 976.
12.7 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. Proceedings of the Fourteenth International Conference On Logic Programming, Leuven, Belgium, pages 286-300. The MIT Press, July 1997.
12.8 H. Simonis and T. Cornelissens. Modelling Producer/Consumer Constraints. In Principles and Practice of Constraint Programming, Cassis, France, September 1995, Springer-Verlag, Lecture Notes in Computer Science 976.
12.9 H. Simonis and A. Aggoun. Search Tree Debugging. Technical Report, COSYTEC SA, October 1997.

13. Using Constraint Visualisation Tools

Helmut Simonis[1], Trijntje Cornelissens[2], Véronique Dumortier[2], Giovanni Fabris[3], F. Nanni[3], and Adriano Tirabosco[3]

[1] COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex, France
email: Helmut.Simonis@cosytec.com
[2] OM Partners
Michielssendreef 42
B-2930 Brasschaat, Belgium
email: {tcornelissens, vdumortier}@ompartners.com
[3] ICON s.r.l.
Viale dell'Industria 21
I-37121 Verona, Italy
email: {gfabris, nanni, tirabosco}@icon.it

In this chapter we describe some experiences with using constraint visualisation tools developed in the DiSCiPl project for the CHIP constraint programming system. We will first discuss their use on some standard examples where we can see how constraint visualisation can be used to detect performance problems and which type of improvements we can suggest. In a second part, we give an overview of some industrial CHIP applications, where the visualisation tools led to significant improvements in the applications. At the end, we present an analysis of the existing tools and some directions for further improvements.

13.1 Introduction

The material in this chapter is derived from a tutorial on constraint visualisation [13.14] developed at COSYTEC and from the DiSCiPl assessment report [13.8]. The visualisation tutorial is a WEB based resource, which discusses several constraint problems and their solution in CHIP. For each problem, alternative solutions are studied using the visualisation tools and an explanation of the observed behaviour is attempted. The assessment report gives a user's point of view on the visualisation tools developed by COSYTEC and PrologIA during the project. It discusses problems of usability and of usefulness of the tools, studied on some example applications.
The debugging tools provide many different views and abstractions of the program behaviour. We try here to discuss how this information can be used to improve the program. This is a first step to a methodology of performance debugging, an area recognised as "a very relevant problem" in the initial DiSCiPl report on debugging needs [13.10]. This chapter only studies the use of the tools, and not the tools themselves. Other chapters of this book contain detailed descriptions of the tools (see chapters 7 and 12). We will concentrate

P. Deransart et al. (Eds.): DiSCiPl, LNCS 1870, pp. 321–356, 2000.

on the use of the CHIP tools in this chapter, a discussion of PrologIA's tools (see also chapter 6) is found in the DiSCiPl assessment report [13.8].

The chapter has the following structure: In section 13.2, we show how the visualisation tools can be used for performance debugging, improving a constraint program by adding redundant constraints or changing the search strategy. We look at five typical problems encountered in some example applications and see what information we can deduce from the visualisation tools. In section 13.3, we look at four industrial applications that where analysed using the visualisation tools. In each case, the visualisation suggested some improvements that had not been seen before. In the last section 13.4, we evaluate the results and suggest possible further improvements of the tools.

13.2 Debugging Scenarios

Developing constraint programs is still more of a craft than an engineering technique. It is quite easy to come up with an initial model which describes a set of constraints for the problem and which uses a standard search method. But that is not enough. Often, this initial model does not work fast enough, or does not find the solution (if it exists) at all. We then enter the domain of performance debugging. There are typically three approaches to improving the program:

- We can think of a new model, expressing the constraints in a different way. This may require a different encoding of the decision variables, or a new way of stating constraints. A typical example is the use of the dual model, which exchanges the roles of variables and values.
- We can add redundant constraints to the model. While these constraints are logically implied by the existing constraints, their propagation may further reduce the domains and allow us to find the solution more rapidly.
- The third and probably most common way is to change the search strategy. Instead of the standard search method, we can try out strategies and heuristics which are problem specific or which were found working for a similar problem. We might also use a partial search strategy rather than a complete one [13.2].

In each case, it is important to analyse the results of the programs and to compare their search spaces. Usually, this is done either by checking the execution time or by counting the backtracking steps. But these are very crude measures of performance and do not tell us much about what is really happening in each particular application. With the visualisation tools, we can gain more understanding by looking for information at a much more detailed level.

In this section we try to cover typical situations which we have encountered in many applications. We will describe five scenarios, each concentrating on one aspect of performance debugging:

– finding new variable/value orderings
– comparing two heuristics
– adding redundant constraints
– discovering structure in the problem
– identifying weak propagation

Note that in large-scale problems these issues do not arise independently, but must be treated together at the same time.

As examples we use classical CHIP demo problems, the N-queens problem [13.15], a map colouring problem [13.11] [13.15], the ship loading example [13.12] [13.1] and the square placement problem [13.1] [13.4]. All of them have been solved before the visualisation tools were available. But in each case, we have found new improvements when we re-considered the programs with our new tools. For space reasons, we can here only concentrate on some aspects of these improvements. A more detailed description of the problems and the different constraint models is found in [13.14].

13.2.1 Finding New Variable/Value Orderings

When we develop a constraint application, the constraint model always is particular to the problem at hand, since normally the constraints are directly derived from the problem. For the search routine, on the other hand, we often start with a standard built-in strategy. A typical example would be to use the built-in `labeling/4` routine which either takes the variables in the input order or selects them according to a pre-defined selection strategy like *first fail*, and which uses the standard `indomain/1` enumeration trying the values in the domain in increasing order. Quite often, this type of strategy will lead to a solution (after some search perhaps), but for some problems, it may not work at all. We will now see how we can use the search-tree tool (see chapter 7) to find new variable and value selection strategies. These strategies will be more adapted to the particular problem we want to solve and can take some domain knowledge into account.
As an example, we consider the classical N-queens problem with a standard labelling routine of *first fail* variable selection and an `indomain` value selection. The CHIP program for this problem is shown below with the annotation required for the visualisation tools:

```
?-lib search.
?-lib visualize.

top:-
        solve(40).
```

```
solve(N):-
        length(L, N),
        L :: 1..N,
        create_dif(L, 1, N, L1, L2, K),
        alldifferent(L),
        alldifferent(L1),
        alldifferent(L2),
        visualize(L,visualize_variable,
                  [winw<-500,winh<-500],queen1),
        search_start(L, label(K), [winw<-760,winh<-500]).

create_dif([], N, M, [], [], []).
create_dif([H|T], N, M, [H+N|R], [H+M|S], [t(H,N)|V]):-
        N1 is N+1,
        M1 is M-1,
        create_dif(T, N1, M1, R, S, V).

label([]).
label([H|T]) :-
        delete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label(Rest).
```

If we test this program for increasing values of N, we can observe the following behaviour. In figure 13.1 we plot the number of backtracking steps needed to find a solution over the increasing board size from 4 to 200. The output is obtained by one of the visualiser tools for CHIP, the `visualize_measurement` tool which displays the number of backtracking steps. We stop the search if we do not find a solution within 5000 backtracking steps. For small sizes of N, we find solutions quite rapidly, with some exceptions which need up to 2000 backtracking steps. But for larger values of N, we quite often do not find a solution within the given limit. Clearly, this default search method is not appropriate for large problem instances.

To analyse the problem more closely, we look at the search-tree generated for problem size 40 and select the node of the first failure (figure 13.2).

We see that failure occurs rather late, when nearly 90 % of all variables have been assigned. We can also see that we only backtrack for a few levels before we correct the problem and find a solution. In the domain state view on the right we see that most variables are already assigned to values. Each line of the display represents one variable, fixed values are shown in light gray with the currently assigned variable shown in black, the domains of the remaining variables are shown in dark gray and the constraint propagation shows crossed out all values which are removed in this assignment step. Note that some col-or transformation was performed on the pictures to make the

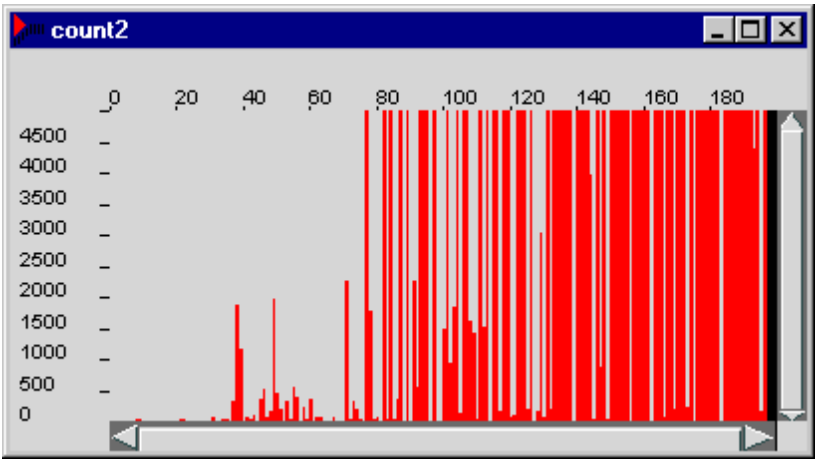


Fig. 13.1. Counting backtracking steps for first fail

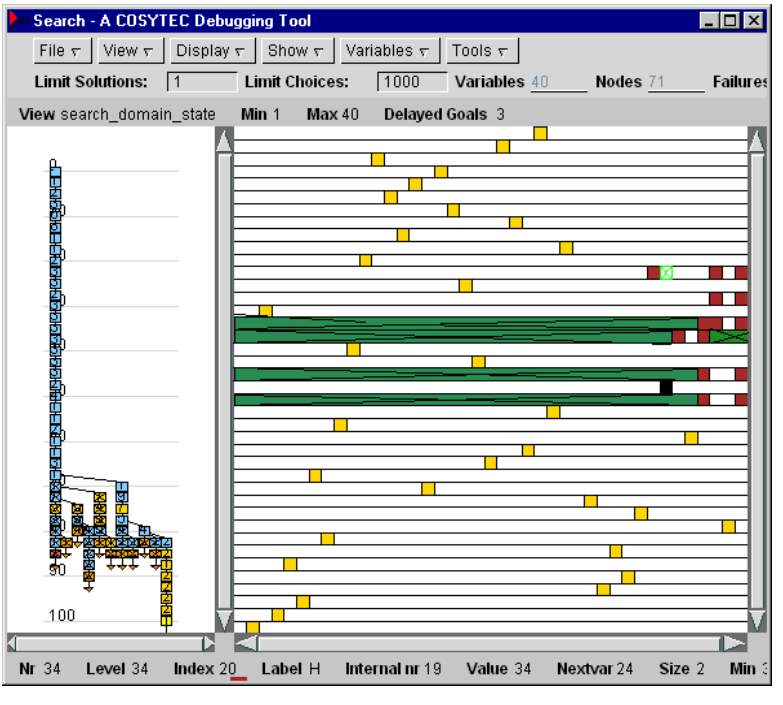


Fig. 13.2. 40-Queens problem, domain state at first failure

information appear more clearly in black and white. Looking at this display, we can make three observations:

– All remaining unassigned variables are in the middle of the board. The *first fail* strategy did select the variables in a dynamic order. The diagram in figure 13.3 shows the variable selection order in more detail.
– All remaining unassigned values are at the top of the domain. The smaller values have been used up, so that only the larger values remain. This is not surprising, the `indomain` value choice will always try the small values before selecting a large value.
– If we count the number of remaining variables and the number of remaining values, we note that there are 6 remaining variables, but only 5 remaining values. Obviously, there can not be any solution at this point, because all variables must have pairwise different values. The forward checking of the `alldifferent` constraint does not detect this. We will come back to this observation later on.

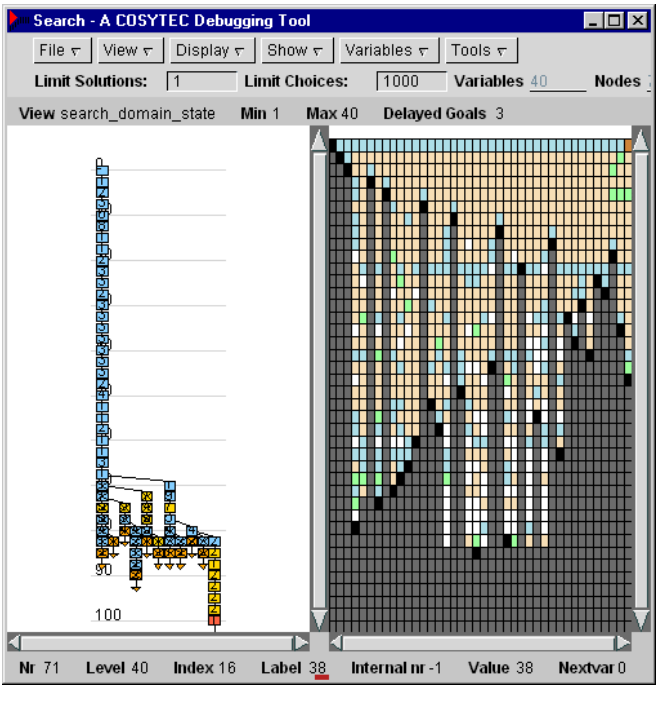


Fig. 13.3. 40-queens, variable selection

In figure 13.3 we see the domain update view. Each line is one assignment step, each column is a variable. The variables in black are assigned at this

step, the variables in dark gray are already fixed. The lighter colours show different types of domain updates. We can see that initially variables on the left of the board are selected, but that the strategy starts picking variables dynamically and that the last variables assigned are in the middle. In the picture we see the path to the first solution, and we can note that when the 7th last variable is chosen, all remaining variables are assigned by propagation.

If we look at the tree for other problem sizes, we observe the same behaviour. The remaining variables are in the centre of the board and the last unassigned values are the high values. This suggests an assignment strategy which tries to change this. If we start assigning the variables in the middle first, then some other variables, perhaps easier to assign, will be left at the end. But as a dynamic variable choice is usually better than a static one, we keep the *first fail* strategy, and just reorganise the variables in a different initial order. For value assignment, we could try and use `indomain(X, max)` which tries to assign values starting from the largest value. But, while this would avoid the problem observed, it would just mean that in the end we are left with unused small values instead. A better choice will be to use `indomain(X, middle)` which starts assigning values in increasing distance from the centre of the domain. This values choice does not have a bias for either small or large values (another idea would be to use a randomised value selection). Implementing this new strategy, we find the following search-tree (figure 13.4):

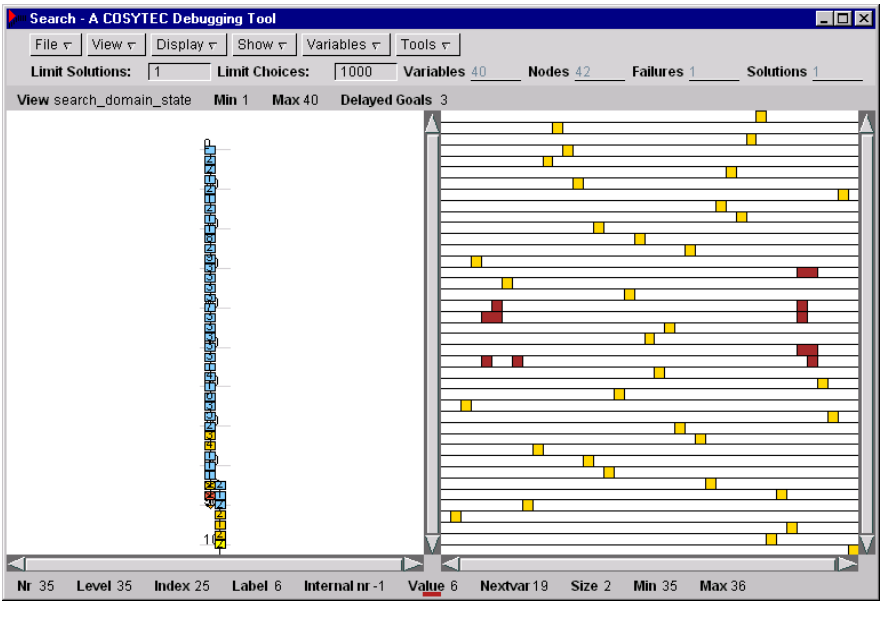


Fig. 13.4. 40-queens, center labelling, first failure

We can see that the search-tree is smaller for this problem, there is only one backtracking step. Selecting the point of the first failure, the last remaining values are both large and small values, all values in the middle have been taken. The last unassigned variables are still in the middle of the board, even though we started the assignment there. But clearly, one problem instance is not enough to decide if this new method is an improvement or not. If we again run all problem instances from 4 to 200, we obtain figure 13.5.

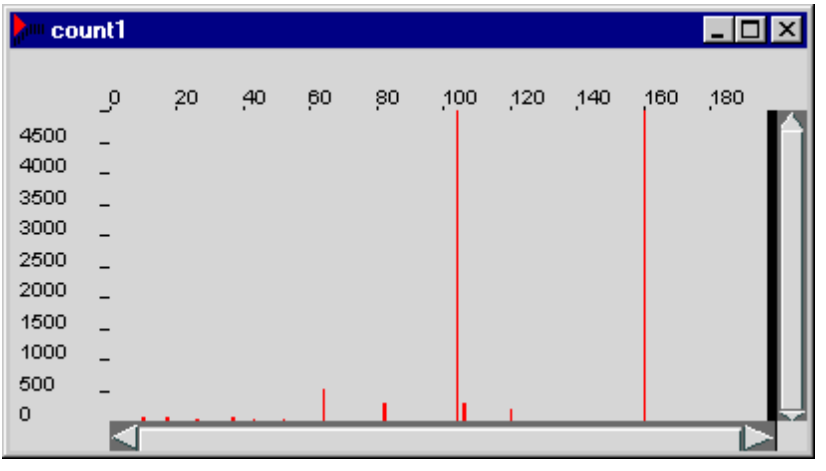


Fig. 13.5. center labelling, all problem instances 4 to 200

Clearly, this is a much improved search strategy. There are only two problem instances where the system does not find the solution within 5000 backtracking steps, and usually it requires a lot less choices than before. If we want to refine the solution to avoid the two exceptional problem instances, we have to introduce partial search techniques, *credit based search* [13.2] works very well for this problem.
How general is this technique to deduce heuristics from the search-tree and domain information? In [13.13], we described a similar analysis for the Mystery shopper problem [13.9], where a different value choice strategy suggested by a failure analysis leads to a no-backtrack solution for CHIP, while with the standard search strategy no solution could be found.

13.2.2 Comparing Two Heuristics

Quite often, in developing search strategies we can come up with two rather similar variants of a program. It is interesting to see the difference between these variants, not only in the final solution, but also in the deduction steps which are used. As an example we can compare two variants of the *first*

fail heuristic for the N-queens problem. The first variant uses the `delete/5` primitive in a recursive user-defined search procedure, the other uses the `newdelete/5` primitive. Both functions select from a list of variables the first variable which has the smallest domain and also return the remaining elements of the list. The difference lies in the way the list of the remaining elements is constructed. `newdelete` returns the list in input order, while `delete` reorders the list, which is somewhat faster. This means that at some later point in the search `delete` may pick a different candidate than `newdelete`. This slight variation can have a significant impact on the search. To visualise the difference, we generate a search-tree which shows both strategies next to each other. This can be done with the code shown below.

```
compare(K):-
        once(label(K)),
        fail.
compare(K):-
        once(label1(K)),
        fail.

label([]).
label([H|T]) :-
        delete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label(Rest).

label1([]).
label1([H|T]) :-
        newdelete(Var, [H|T], Rest, 1, first_fail),
        Var=t(X,N),
        search_node(X,N,indomain(X)),
        label1(Rest).
```

Figure 13.6 shows the resulting trees for the 40 queens problem, on the left is the selection with `delete`, on the right the selection with `newdelete`.

Superficially, the trees look quite similar, but checking the selected variable at each step shows some difference rather early on. This difference becomes more clear with a *phase-line* display (figure 13.7), which links all nodes in the tree which select the same variable. This views indicates when some variable is assigned in different parts of the search-tree.

Figure 13.7 shows that initially, the same variables are selected in both trees, but that then the order of selection differs significantly. We can also see that even for one strategy the selection is not always the same, on backtracking other variables are more constrained and therefore selected first. The diagrams above have shown that the search-tree and the selection for the two

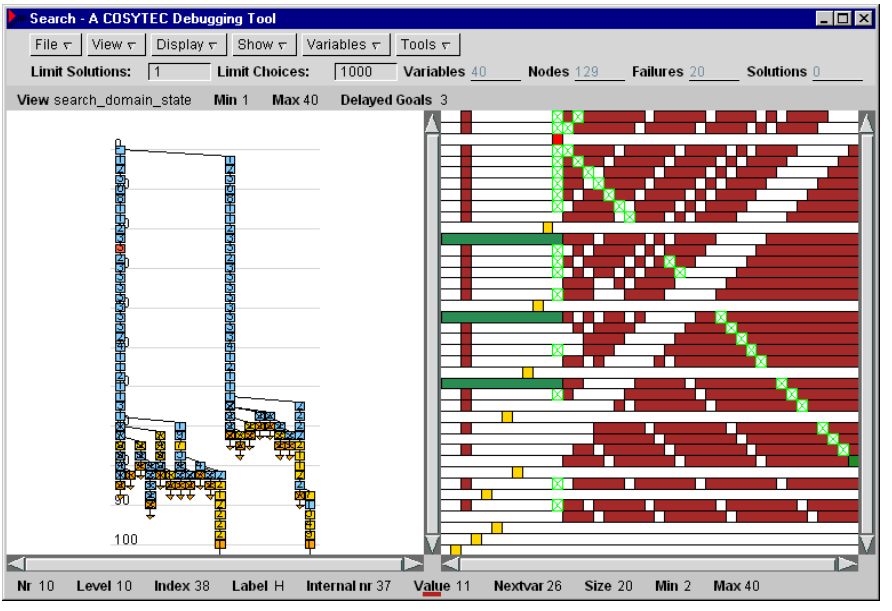


Fig. 13.6. Comparing strategies

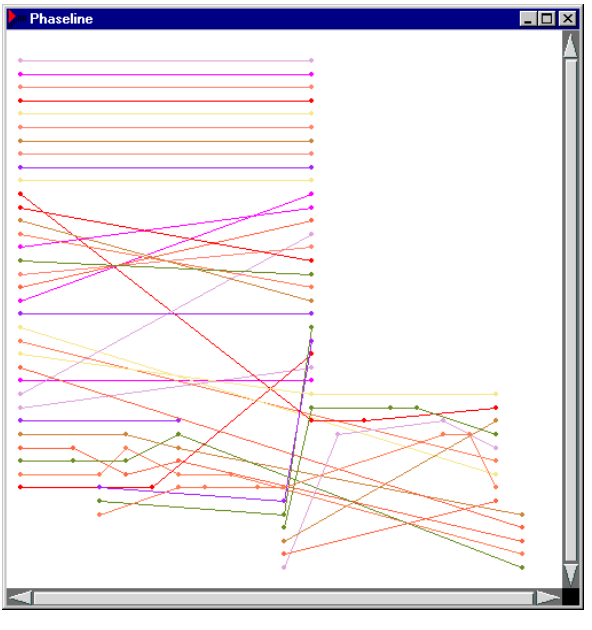


Fig. 13.7. Phase line display

very similar search strategies vary even for the quite simple N-queens problem. But perhaps this is just a matter of performing the same steps in a different order, and the resulting solutions are quite similar? Figure 13.8 below demonstrates that this is not the case. The figure shows an overlay of the two first solutions that are found. Queens in black are assigned in the common part of the search-tree, the gray coloured queens show the assignment where the two solutions differ, there are only two queens outlined in black which are placed in both solutions to the same location after the search-trees diverge. The small difference in the selection function leads to two quite different solutions. This technique of comparing two search-trees can be applied in many other situations, for example to check the impact of some redundant constraint, or the effect of some small change in the input data. The *phase-line* display allows to identify quickly if there are major differences in the variable order, it is also possible to identify *isomorphic sub-trees* (see chapter 7).

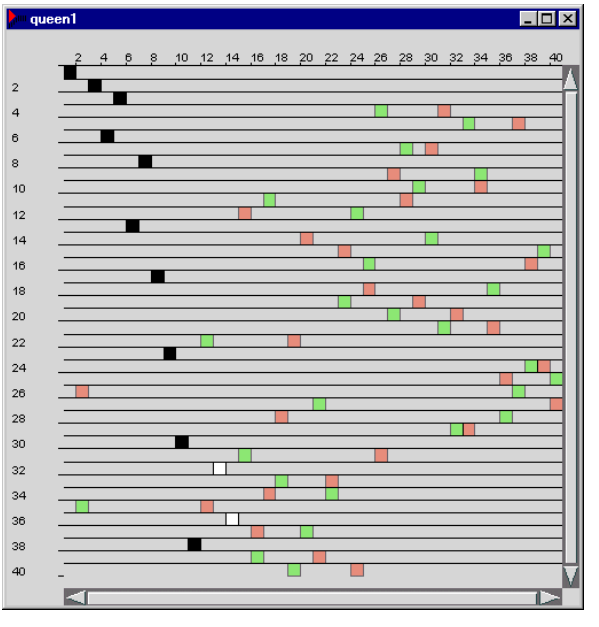


Fig. 13.8. Overlay of solutions

13.2.3 Adding Redundant Constraints

In many constraint programs, the results can be significantly improved by adding redundant constraints, i.e. constraints which are logically implied by the original constraints, and which are only added to improve propagation.

Obviously, this improved propagation must be balanced against the cost of processing the new constraints. To understand the value of a particular redundant constraint, it is useful to see the effect on the search-tree and on the variable domains. As an example, we choose the ship loading problem from [13.12], which is used for one of the standard CHIP examples [13.1]. This problem is a simple resource restricted scheduling problem, where tasks, linked by inequality constraints, are also requiring different amount of manpower resources, with an overall limit on the available manpower at each time point. This problem is modelled with inequality constraints and a `cumulative` [13.1] constraint in CHIP. But there is another possible, redundant constraint that can be used, the `precedence` constraint [13.3]. In figures 13.9 and 13.10, we show the search-tree without and with the `precedence` constraint. Note that the tree without the redundant constraint has 470 nodes, and the tree with the redundant constraint only 287 nodes. But due to the dynamic variable selection, the trees are not directly comparable, as the variables are assigned in a different order.

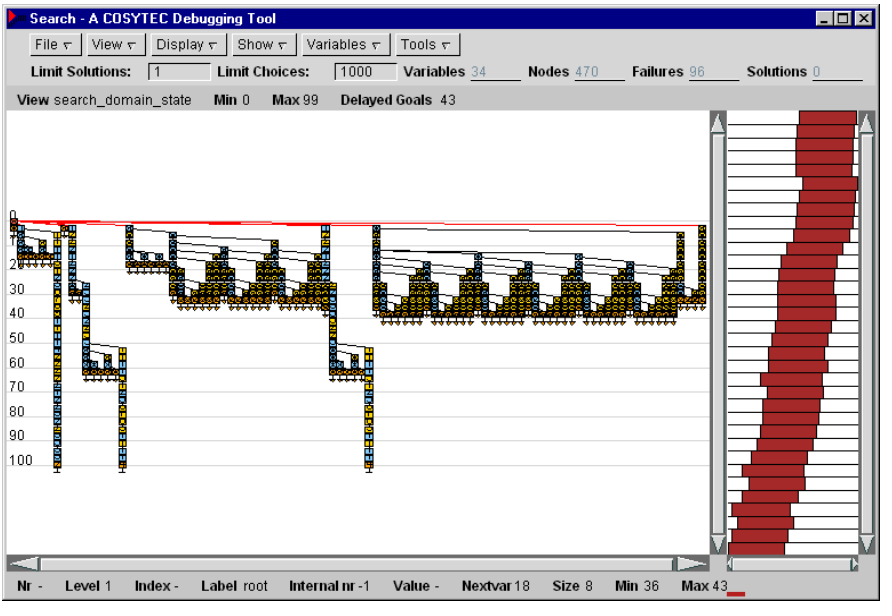


Fig. 13.9. search-tree without redundant constraint

We can also compare the domains of all variables after the constraint setup, before the search starts. The first image (figure 13.11) shows the domains without the added redundant constraint, the second one (figure 13.12) shows the domains with the added redundant constraint. It is clear that the domains are much reduced.

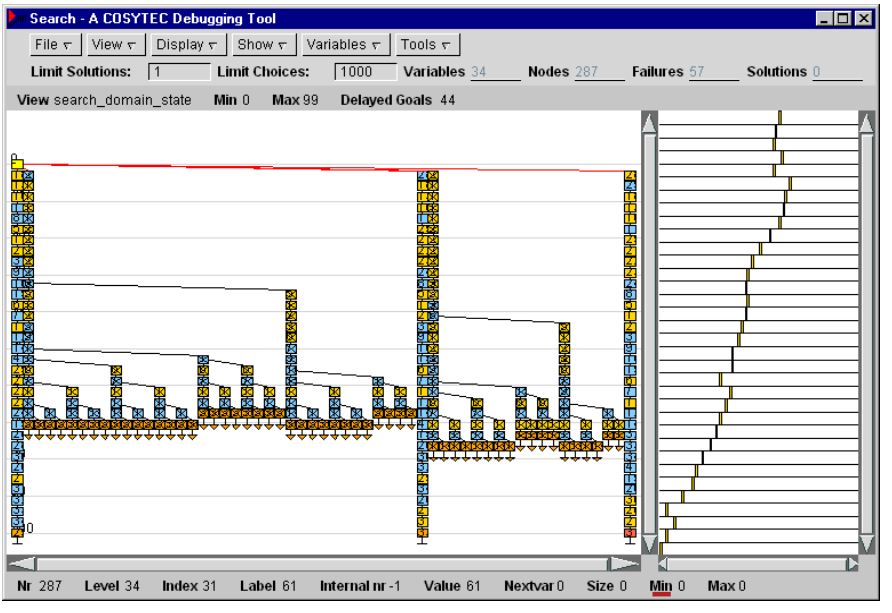


Fig. 13.10. search-tree with redundant constraint

Our visualisation tools currently do not provide any way to compare run times of different strategies or constraints. Due to the instrumentation of the solver generating the tree, this would not show the real execution times in any case. To measure execution times, it is still required to run both programs

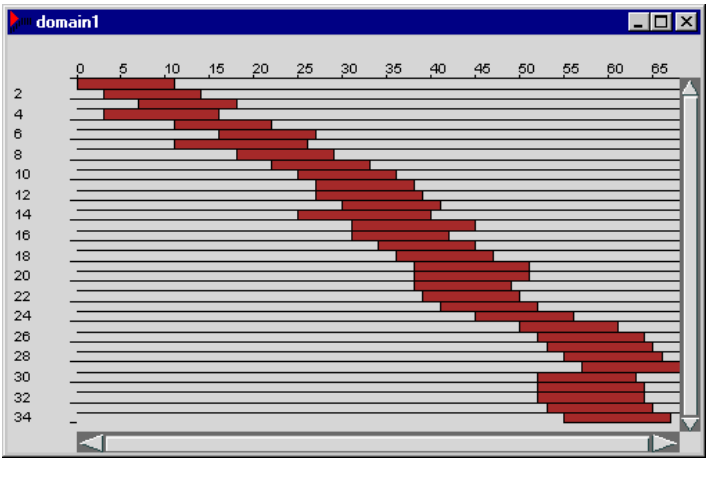


Fig. 13.11. Initial domains without redundant constraint

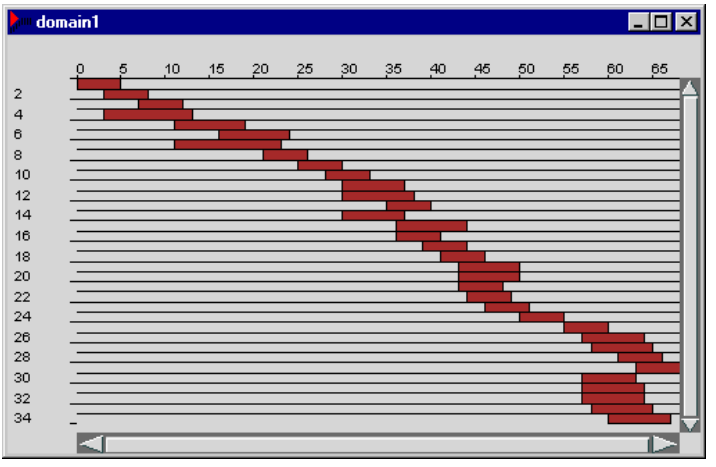


Fig. 13.12. Initial domains with redundant constraint

with the same data-sets and compare overall results. The search-tree and visualisation tools at the moment only allow us to get a rough idea of the value of redundant constraints. We can make a more detailed analysis looking at the *propagation events* (see chapter 7) for each constraint, seeing how often a constraint is woken, how often it detects failure or domain reductions. This also applies to global constraints, which always use a combination of different methods to express the declarative constraint. With the propagation events, we can see which methods are used for which type of problem, and which methods are just overhead for some problem instance.

13.2.4 Discovering Structure in the Problem

It is often useful to search for some hidden structure in a problem, which we can exploit in a search method. One useful tool for this purpose is the constraint *incidence matrix* in the search-tree tool, which shows all variables on the x-axis against all constraints on the y-axis. Figures 13.13 and 13.14 are the incidence matrices of the same problem, but with different initial variable orderings.

The problem we study here is the map colouring problem for the 110 country map from M. Gardner, another classical CHIP example program [13.15][13.11]. The figures show that the variable order given by M. Gardner (figure 13.13) contains a lot of structure, which makes it a good candidate for a static variable selection order. In the randomised order (figure 13.14), all appearance of order is gone, we would have to discover this order again with our search routine to obtain good results.

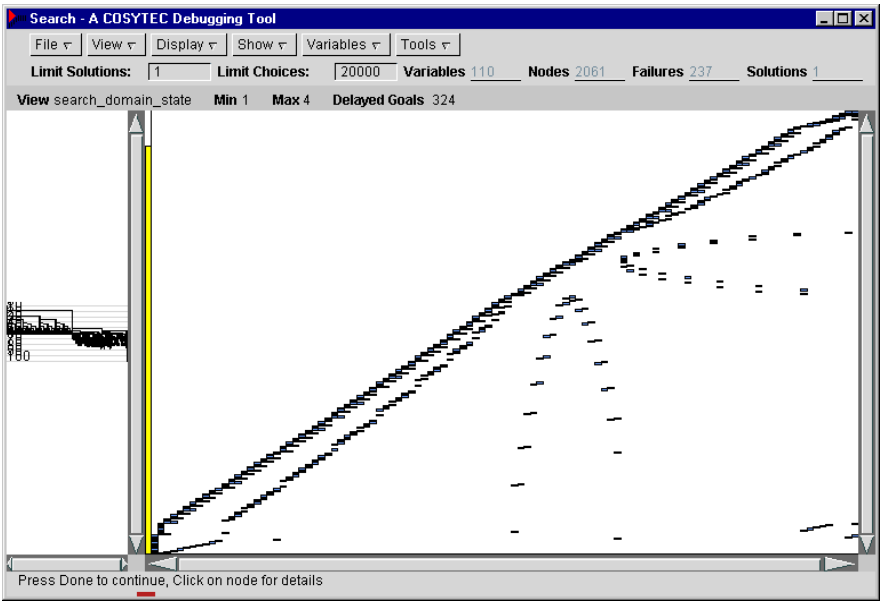


Fig. 13.13. Incidence matrix with specific variable ordering

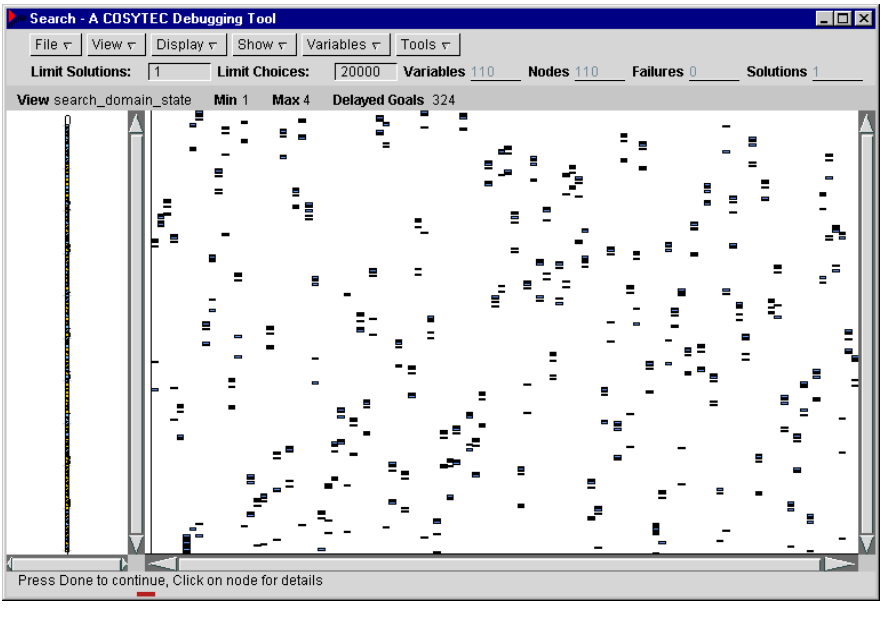


Fig. 13.14. Incidence matrix with random ordering

13.2.5 Identifying Weak Propagation

The last example for the use of the search-tree visualisation is the discovery of weak propagation in a program. We had already mentioned in the first example of N-queens that we can find the cause of the failure in the initial search-tree by counting the variables and possible values remaining. This is a typical situation where we analyse a state of the computation and check manually if there is any missing propagation. Obviously, this requires a good understanding of the problem and of possible propagation methods, even those that are not currently used in the constraint engine. We can try to automate this reasoning by applying a general propagation mechanism to detect if some values can be removed in the domains. *Shaving* [13.7] is such a general method. After each propagation step, we test each value in the domain of each variable to check if it can still be assigned. If the test assignment fails, we can remove the value from the domain before continuing the search. It is interesting to see if we can come up with a more specific deduction rule that would allow us to remove the shaved values without the expensive testing at each step. Below we show two shaving examples from the square placement problem [13.1][13.4]. It overlays the domain display with and without shaving at two steps of the search. The dark coloured areas are values which stay in the domain, the lighter coloured values are removed by shaving. In the first example (figure 13.15), the domain reduction is quite insignificant, but in the second step (figure 13.16) a bit deeper in the search, many values are removed and some additional assignments are detected.

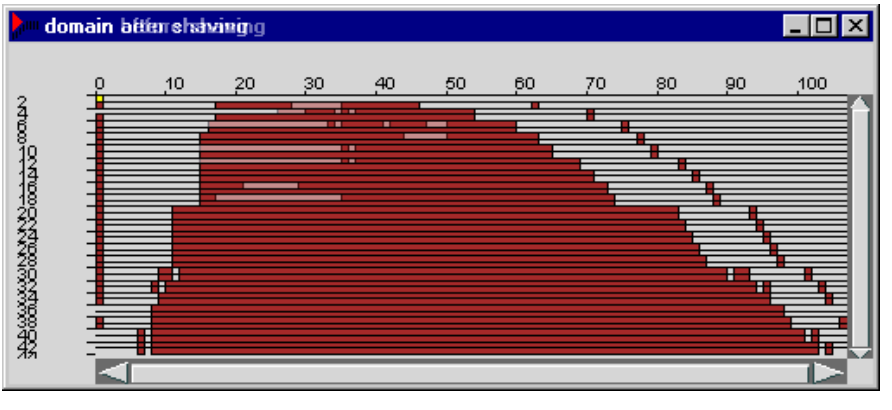


Fig. 13.15. Shaving at the first step

We may not be able to deduce any new deduction rules, or we may not want to spend the time doing this for some particular problem. In that case we can still apply the shaving directly, if the additional domain reductions balance the cost of the testing operations.

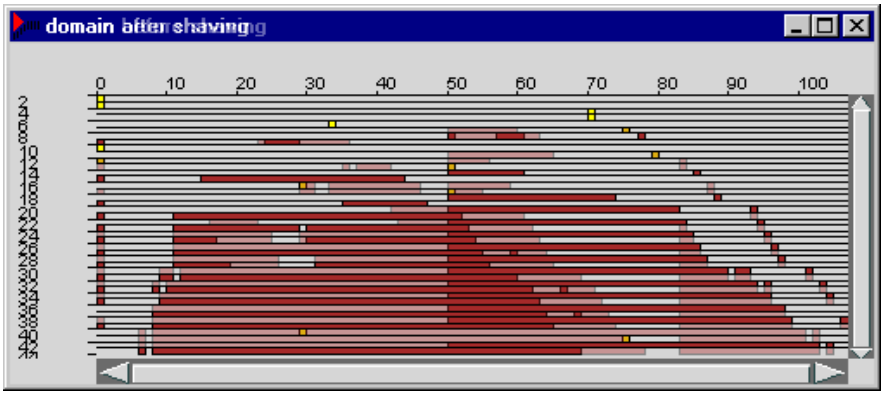


Fig. 13.16. Shaving operation removing many values

13.3 Case Studies from Industry

This section describes a number of industrial applications that were checked with the visualisation tools for CHIP. The applications had been developed and sometimes delivered to the customers before the search-tree tools had been available. As part of the DiSCiPl assessment phase, the applications were analysed with the tools to discover the usability of the tools, but also to see if improvements to the original programs could be made. The first two applications have been developed by OM Partners, the last two by ICON. More details about the applications and the assessment results can be found in [13.8].

13.3.1 Scheduling Application Involving Setup Cost

Problem Description. The problem consists of scheduling a number of tasks (about 100 tasks) on 8 machines. Machines are divided in two classes. Each task has to be scheduled on a machine in a given class, depending on the task type. Moreover, the machines allowed for a task are ordered by preference. Some tasks are already fixed on a particular machine. Most of the tasks additionally use a common resource (laboratory personnel). The number of instances for this resource is limited. Further, each task has a different *bill of material* and the availability of raw material is limited. The schedule also takes into account the unavailability of machines and resources. The end products produced by the tasks belong to different product families. Transitions between different product families in the schedule involve a setup cost. For each task, an earliest and latest production time is specified. The earliest time is a hard constraint, the latest time can be relaxed. The scheduling horizon is 0..50000 minutes (about 34 days). The goal is to find a schedule that minimises the overall setup cost while at the same time keeping the lateness

of tasks under control, also taking into account the preference of machines for each task. A parameter allows weighting the importance of setup cost against the lateness of tasks. As the search space is quite large, the problem is relaxed to finding a good quality solution (sub-optimal) reasonably fast (i.e. within some seconds).
The program already existed before the start of the DiSCiPl project. It includes several `diffn` and `cumulative` constraints (besides equations, inequalities and `element` constraints). A sophisticated labelling strategy is used to find a reasonably good solution. At the time the program was written, the `cycle` constraint was not yet available. Therefore, reducing setup costs was integrated in the labelling strategy. In the context of DiSCiPl, the program was rewritten using the `cycle` constraint. This constraint allows modelling the sequence of tasks on each machine as a cycle, in which the setup costs act as weights.

Evaluation. We use the visualisers `visualize_cumulative_resource` and `visualize_assignment`, combined with the search-tree tool showing the end time and machine variable for each task. In the new version of the program, `visualize_graph_lines` was added to visualise the `cycle` constraint. Adding the necessary annotations for the search-tree tool and visualisers required only minor modifications to the program. In order to use the visualisers, `visualize/4` wrappers are simply put around the `cumulative`, `diffn` and `cycle` constraints. However, some care is needed when the same routine setting up a constraint is called several times (e.g. the same routine may be used to set up a `cumulative` constraint for each resource). In that case, a unique identifier has to be created for each visualiser (based on the information passed to the routine). Using the search-tree tool requires to create the list of search variables and to store some bookkeeping information (in order to obtain the number of the search variable during the labelling phase). Adding the latter information is very easy if the program uses CHIP++ objects. Each task is modelled as an object instance, containing all information about this task. The number of each task for use in the search-tree tool can just be added as an extra data field in the object instance.
Before labelling is started (i.e. just after setup of the constraints), the visualisers already show the position and resource use of fixed tasks (tasks that cannot be moved and dummy tasks used to model unavailability of machines/resources). This allows verifying the correct modelling of fixed parts. Afterwards, one can check how labelling proceeds. For example, the debugging tools allow checking if tasks are considered in the correct order, e.g. if ordering is based on minimal end time. However, when ordering is based on more complex criteria (e.g. setup costs between task types), the tools do not provide much help. One still has to resort to more complex (external) visualisation tools (e.g. a GANTT representation where tasks can be coloured according to task type). By default, the displayed area in the visualisers is limited by the upper domain limits of the associated variables. This yields

a general overview of the problem. But, it is also possible to zoom in on a specific area in order to obtain more details.

The search-tree tool confirms that, in the original program version (search-tree shown in figure 13.17), the implemented labelling strategy yields a reasonably good solution without backtracking (setup cost: 44, cost of tasks being too late: 3216 ⇒ overall cost of 3656). However, the search space is too large to get to the optimal solution. Use of `min_max` with the time-out parameter to limit the search resulted in a (sub-optimal) solution with cost 3390 (setup cost: 26, cost of tasks being too late: 3130), for the given data set. As mentioned above, an alternative program was developed that uses a `cycle` constraint. The `cycle` visualiser helped in setting up this new model (e.g. helped in identifying the missing propagation due to errors in the modelling and showed the need to use the `origin` parameter of the constraint). The labelling part could be simplified by exploiting the `cycle`: direct labelling of the weights (representing setup costs between tasks) could be used in order to limit setup costs. The remaining labelling code focuses on restricting lateness costs. For the example datasheet, the new version of the program (search-tree

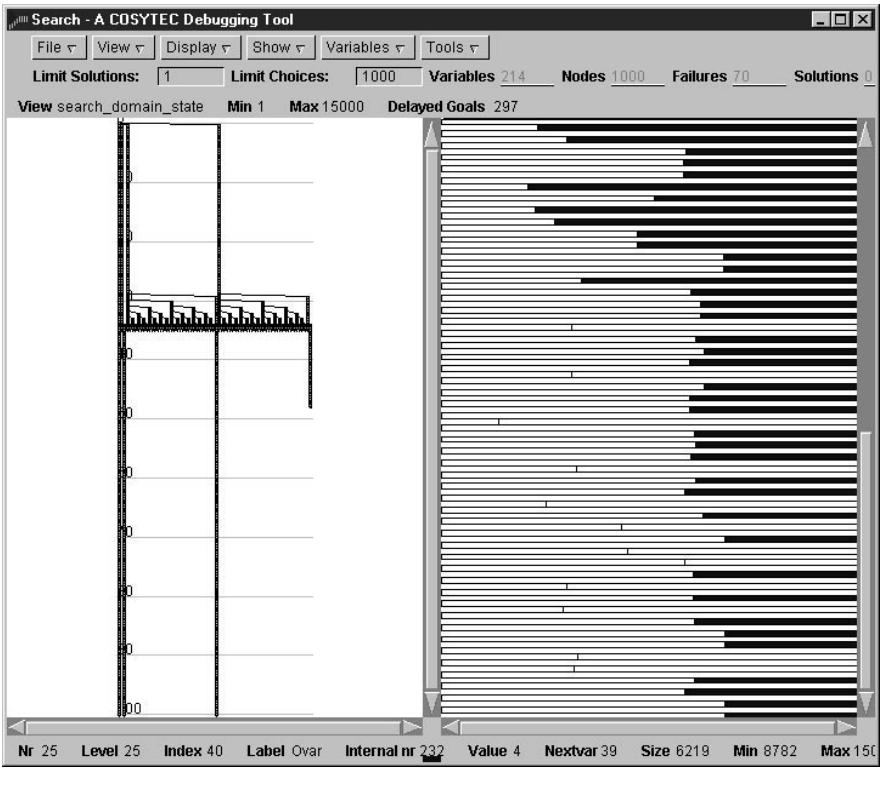


Fig. 13.17. Original program with min_max

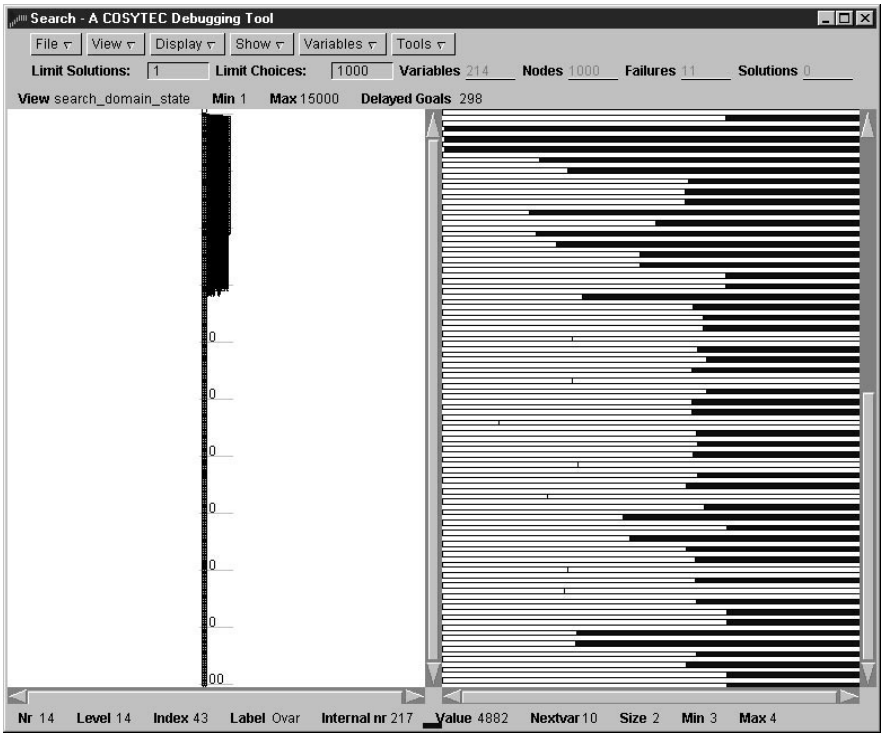


Fig. 13.18. New program with cycle and min_max

shown in figure 13.18) results in a first solution with cost 3494 (setup cost: 18, cost of tasks being too late: 3314). This is significantly better than the one obtained with the original program as far as setup costs are concerned (the `cycle` leads to a better restriction of the search space).

The original program contained an error in the construction of fixed (dummy) tasks to model the unavailable periods of resources, causing no such tasks to be created. The visualisers immediately showed the absence of these tasks.

13.3.2 Tank Scheduling Application

Problem Description. The problem described in this section is a tank planning + shipment problem. A task consists of the following steps: creation of a certain product quantity in a reactor, storage of the product within a tank, and loading (shipment) of the product out of the tank using some loading machine. As soon as the reactor step is finished, the product has to be put into some tank (fill action). This tank is chosen among a pool of available tanks. Of course, tanks have a limited size and a tank should not contain different products at the same time. At some point, (a part of) the tank content

has to be loaded (empty action). The choice of loading machine depends on the tank. Loading is only possible during working hours. The problem has been solved in different phases: The first goal was to schedule the fill, empty and loading actions, keeping the reactor steps fixed and taking into account all tank and lateness constraints. The program involves several `cumulative` and `diffn` constraints (on tanks and loaders). Alternative formulations are possible to take into account the tank capacity:

- a `diffn` in 3 dimensions (each item in the `diffn` models the storage of a product in a tank, lasting from fill to empty, after the product has come out of a reactor step), or
- a `cycle` using loading and unloading operations (argument 12 of the `cycle` constraint [13.6]; this `cycle` models the sequencing of fill and empty actions on tanks).

The second goal was to also take the reactor planning into account. On the reactor there are major setup times between different product families. As usual, it is difficult to determine the relative importance of setup time versus lateness.

Evaluation (Part I). In this part a solver has been made for the tank assignment. The reactor steps are fixed.
We use the `visualize_cumulative_resource`, `visualize_assignment` and `visualize_graph_lines` tools together with the search-tree tool showing the start time, end time and machine variable for each task.

The `cycle` visualiser shows how fill and empty steps on the tanks are scheduled. E.g. in figure 13.19 below (note: a fill and its corresponding empty step have subsequent numbers), the sequence on tank 8 (first line in display)

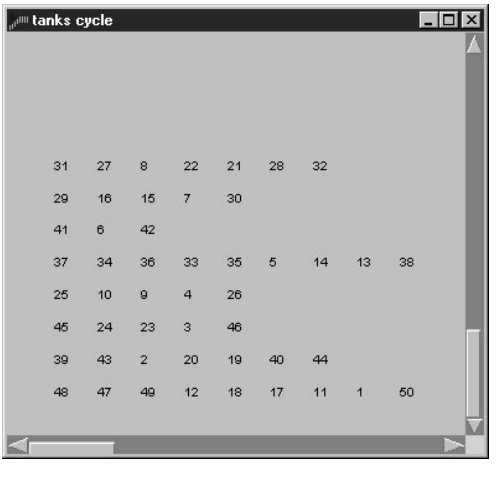


Fig. 13.19. Cycle visualiser on tanks

is fill1(22)-empty1(21)-fill2(28)-fill3(32)-empty3(31)-empty2(27); the tank is empty after step empty1 (21) and after empty2(27). Unfortunately, the step numbers shown in the `cycle` do not reveal which product is involved. So, it is difficult to check whether a tank never contains a product mix. In fact, the program initially contained a bug with respect to this constraint. This bug was only detected through an external visualisation tool (part of our own application) where colouring of tank steps on product type is possible.

The `cycle` visualiser only provides information about the relative sequence of fill/empty steps. It does not indicate their absolute position in time. The latter can be derived from the `diffn` visualiser on tanks. However, a problem is that step numbers do not correspond in the different visualisers (see below). A `cumulative` visualiser shows the workload of the loaders. We can see at what times the maximum capacity is reached. The `diffn` visualiser gives more information about which loaders are used at what times.

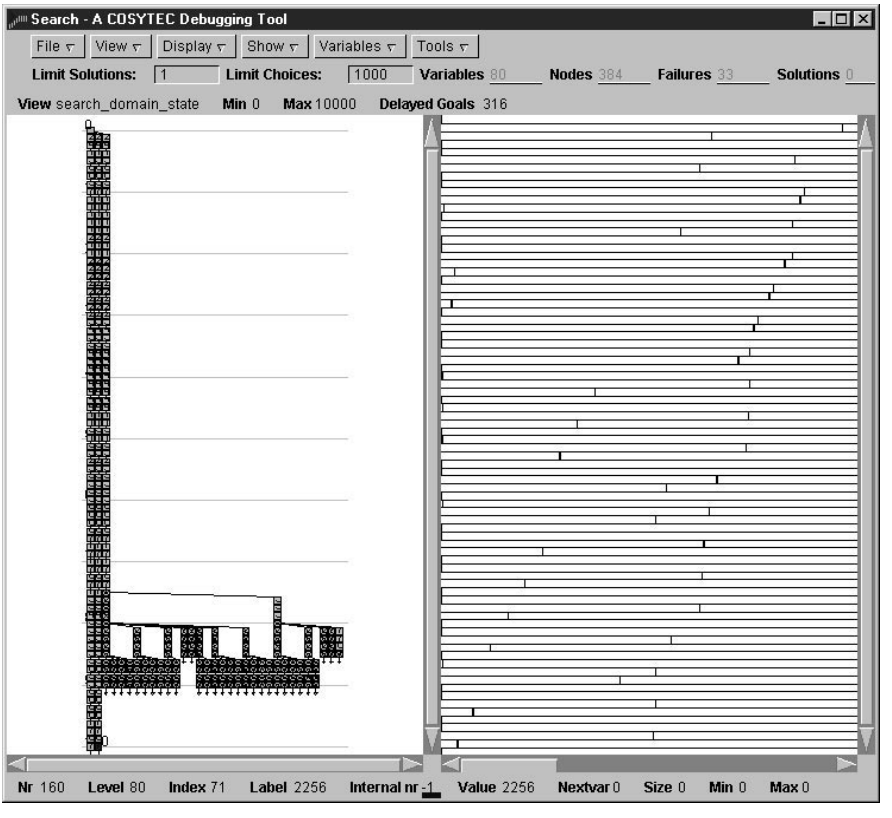


Fig. 13.20. The (incomplete) search-tree view reflecting useless backtracking (trashing)

Evaluation (Part II). The reactor steps have to be scheduled in such a way that the setup time + lateness cost is minimised and that the tank planning (see Part I) is still feasible.

We used additional `visualize_assignment` and `visualize_graph_lines` tools in combination with the search-tree tool.

In this case the `cycle` visualiser and the `diffn` visualiser both show the sequence of the production steps on the reactor. The `diffn` visualiser better reflects the time aspect of the problem, the `cycle` visualiser better reflects the number of tasks already placed. Unfortunately, the steps shown in the visualisers do not reveal which product is involved nor how high the associated setup and lateness cost is. The search-tree view, however, allowed to detect that the node numbers in the `cycle` constraint were not defined correctly. Once the constraints on the reactor scheduling were modelled correctly and the first feasible solution had been found, the search-tree view also was a real help during the optimisation of the solution. The format of the search-tree helped the user to detect trashing and showed at which point the program performed (useless) backtracking (see figure 13.20).

The trashing could be avoided by changing the labelling routine. The users realized that it is not required to try every successor of a node. It was sufficient to try, within the same product family, only the step (i.e. successor) with the earliest due date. The search-tree obtained in the final version of the program is shown in figure 13.21. The ameliorated labelling routine allowed the user to obtain an optimal solution in a reasonable time.

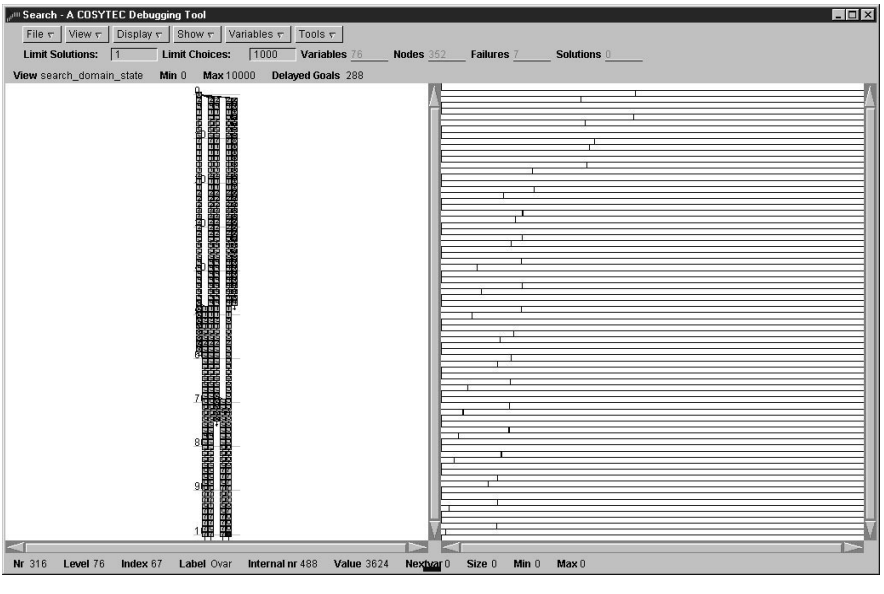


Fig. 13.21. The complete search-tree obtained for the final version of the program

13.3.3 Layout of Mechanical Objects: Graph Colouring

Problem Description. The problem consists in the determination of the configuration of mechanical pieces, which must respect several mutual relations. An instance of the problem is given by:

– a set of objects to configure;
– a set of relations to be satisfied.

Configuring an object means setting up its elements. These elements can be disposed with a set of physical assembly constraints. The relation that must hold between the objects (different for each problem instance) is satisfied by the identification of the elements to be included in any object. The problem is divided in two main parts. This section considers the first one, which consists in the identification of the mechanical elements to be inserted in each object, in order to satisfy the set of relations between them. This problem is basically a *graph colouring* problem with a complex specification, where the graph is defined by the set of relations between the objects. The goal is to find a colouring of the graph that minimises the number of colours used, which is given by a `min_max` minimisation. The domain variable selection strategy is *most constrained*, and the value generation starts from the domain minimum value. The model only uses disequality (`#\=`) constraints. The size of problem instances varies from one to several hundreds of objects and relations. Just as an example of the complexity of the constraint problem, on a small instance with 10 objects and 12 relations, 926 disequality constraints are posted on 65 variables to define the corresponding graph.

Evaluation. For this example, we only used the search-tree tool. There were no major problems annotating the program for the search-tree tool. The `min_max` timeout parameter was included in our production application because an optimal solution can not always be found for each instance of the problem. We saw that with the search-tree tool this parameter is useless, and that it can be replaced by the GUI field `Limit Choices`. Attempting to include in the tree the cost variable (which is not assigned in the labelling) required some extra work on the `search_start` call.

With the tool we found a first benefit without any particular intervention on the code, as two facts were immediately visible: the number of generated constraints and the number of variables. These two values vary a lot depending on the problem instance, and characterise well how hard the problem is. The variables/constraints incidence matrix was very useful in finding disequalities (`#\=`) on variable pairs that were duplicated during the constraint generation phase.

This problem was already recognised during the original code development (of course without the search-tree tool), and is due to the complexity of the generation of constraints for an instance of the problem. The number of duplicated constraints did not seem big enough to significantly influence

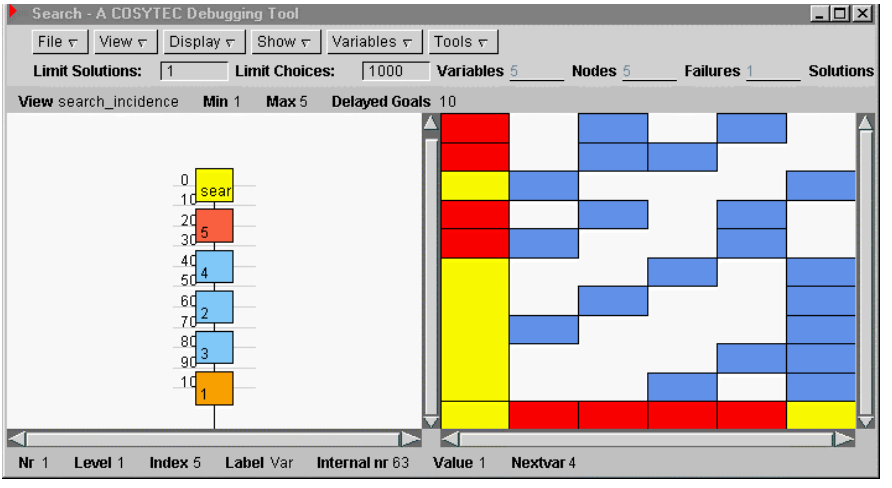


Fig. 13.22. Layout first phase, incidence matrix

the computation; thus this problem was ignored during the development. But this possible optimisation of the program was easily discovered with the search-tree tool (at least for small instances) by checking the incidence matrix (figure 13.22). With the tool we verified that the implemented search strategy is useful to quickly find good solutions. However, even for medium sized problems the optimal solution requires the exploration of wide trees. As an example, the next picture (figure 13.23) shows a tree exceeding over 5000 nodes (restricted with the tool parameter `Limit Choices`).

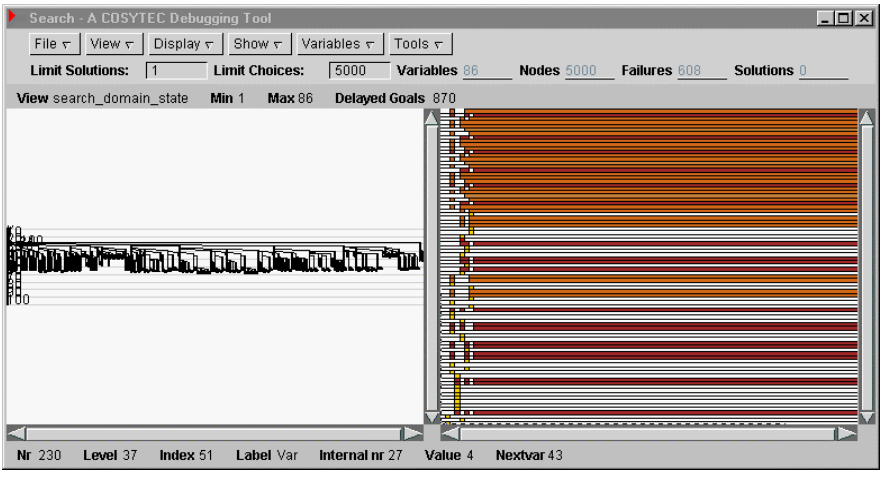


Fig. 13.23. Layout first phase, domain state, limited search

During the analysis of small problems with the search-tree, we discovered that the labelling does not avoid the generation of solutions that are permutations of each other. In order to recognise this behaviour, we manipulated the tree view by collapsing some branches. These branches are equivalent, as after the first domain variable assignment they lead to the same propagation, differing only for a permutation of the assigned values. In figure 13.24, the two solutions found are shown, each with the same value for the first variable. In addition, the first failure branch with the same first variable (index 39, value 1) is expanded. The search for a solution could stop even after the first failure branch exploration, as it represents the optimality proof of the previous solution: the following sub trees are equal to the expanded one, except for the assignment order; for instance, the next branch starts with the variable 39, value 2.

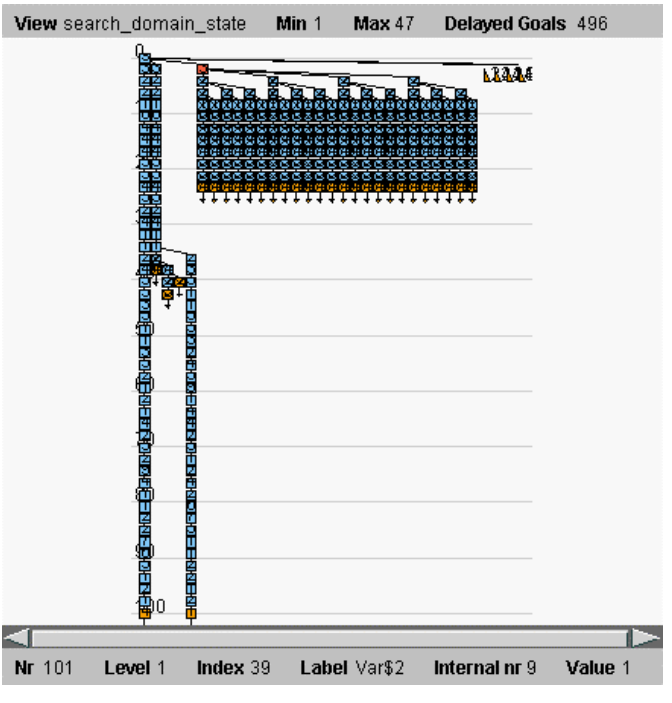


Fig. 13.24. Layout first phase, first solutions found

This problem was considered even in the development phase, but the need for a quick near-optimal solution led to a timeout-limited search without making the generation function more complex. The actual generation is based on the selection of most constrained variable, which is given the minimum possible value with `indomain/1`.

```
...
search_start(Ls,min_max(outlab_stv(Ls,Cost),
                        Ls, 0, Num, 0)),
...

outlab_stv(Ls,Cost):-
        maximum(Cost, Ls),
        search_number(Ls,Merge),
        labeling(Merge, 1, most_constrained, assign_stv).

assign_stv(t(X,N)):-
        search_node(X,N,indomain(X)).
```

On average this strategy seems to lead quickly to a solution that uses few colours (therefore frequently optimal, even if the proof requires a deeper exploration of the search-tree). We modified the generation function to avoid the permutation effect described above. The labelling function then became:

```
...
search_start(Ls,min_max(delete_new(Ls,Cost,[0]),
                        Ls, 0, Num,0))
...

delete_new([],_,_).
delete_new([H|T],Cost,CostP):-
        maximum(Cost,[H|T]),
        delete(t(X,N),[H|T],R,1,most_constrained),
        search_node(X,N,assign1(X,CostP,CostPNew)),
        delete_new(R,Cost,CostPNew).

assign1(X,C,C):-
        C\=[0],
        member(X,C).
assign1(X,C,[H1|C]):-
        C=[H|_],
        H1 is H +1,
        X=H1.
```

With the search-tree tool it was then possible to see a reduction in the tree. Unfortunately, we noted that the instances of the problem vary greatly. Consequently, the same generation function does not show a uniform behaviour; i.e. it is not always better than the others, at least when measuring execution time.

13.3.4 Layout of Mechanical Objects: Physical Layout

Problem Description. This section considers the analysis of the second phase of the mechanical object layout application described previously. Given the set of objects to configure, the second part of the problem is aimed at the physical layout of the elements in each object, meeting a fixed set of physical constraints (which are invariant for any instance of the problem). In this phase there is no need of an optimal solution, we are satisfied with the first solution which properly places the elements identified by the first phase. This stage involves the use of a `cumulative` constraint to define the physical constraints that mechanical elements must respect. Other used constraints are `among`, `element` and `alldifferent`.

Evaluation. We used the `visualize_cumulative_resource` tool together with the search-tree tool.

The goal of the physical layout phase is to find a placement for the mechanical elements. For any object there are two `cumulative` constraints that express the physical assembly limitations. Each object element is represented with a task that uses one resource. Its "duration" is the space that could cause a placement conflict if owned by another element. The `among` constraint captures the optional distribution of the elements in particular areas of every object. The first analysis was run on a small example consisting of 7 objects and 7 relations. This way all the visualiser windows could be seen together with the search-tree.

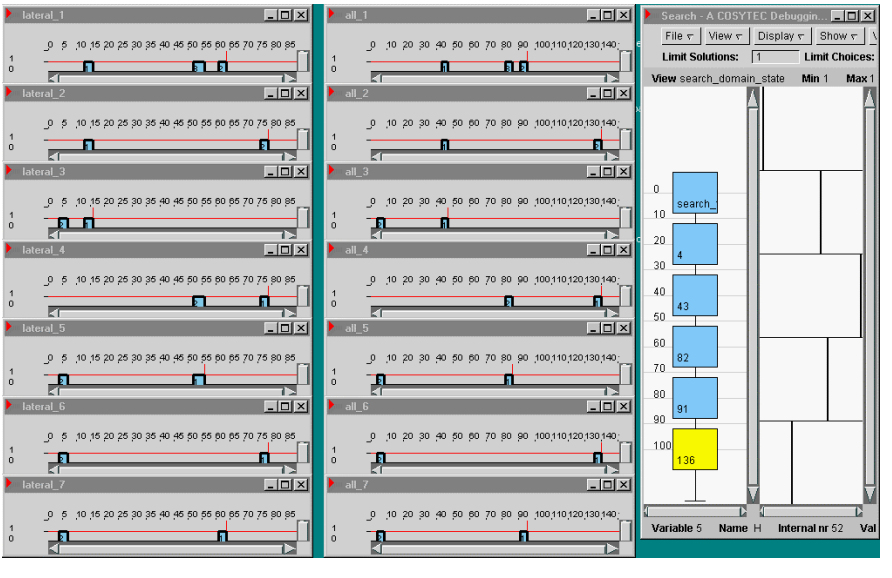


Fig. 13.25. Layout second phase: lateral_x, all_x and search-tree windows

For any object there were two paired windows (named lateral_x and all_x), representing the two physical constraints. As we can see in figure 13.25, there were five elements to place (corresponding to the nodes in the tree) that could be joined differently within the objects. For instance, in the all_1 window there were three elements, whereas in the all_3 window there were two. A graphic representation of the element positions is interesting, even if the `cumulative_resource` representation does not completely match our application domain. We can verify on the search-tree that for small problems there is no backtracking at all when the variables are assigned values that are compatible with the physical constraints and there are no additional `among` constraints. If an optional distribution is added, the original labelling initially leads to assignments that do not satisfy to the added `among` constraint (figure 13.26).

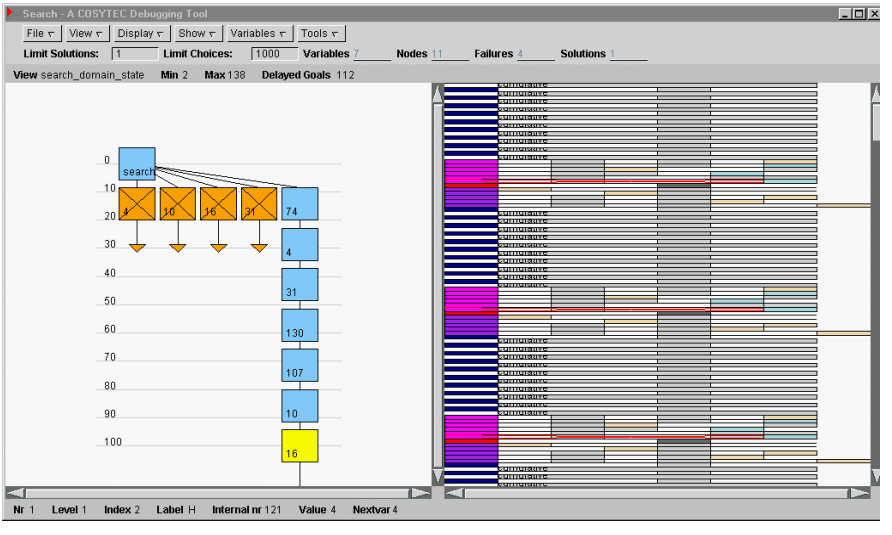


Fig. 13.26. Layout second phase: failures due to among

In order to detect the reason for these failures, the propagation view is quite helpful. For many problem instances we can see that there is no solution at all, due to the extra constraint `among`. For a larger data-set, we could not identify the reasons for failure, mainly due to the large number of constraints to visualise and the difficulty of selecting a good subset to analyse. On smaller problem instances, a detailed analysis of the propagation view proved to be useful, since it revealed a large number of inappropriate constraint calls and useless propagation (see figure 13.27):

With the incidence matrix view (figure 13.28), we could find the excessive number of `element` constraints. It is worth noting that the piece of

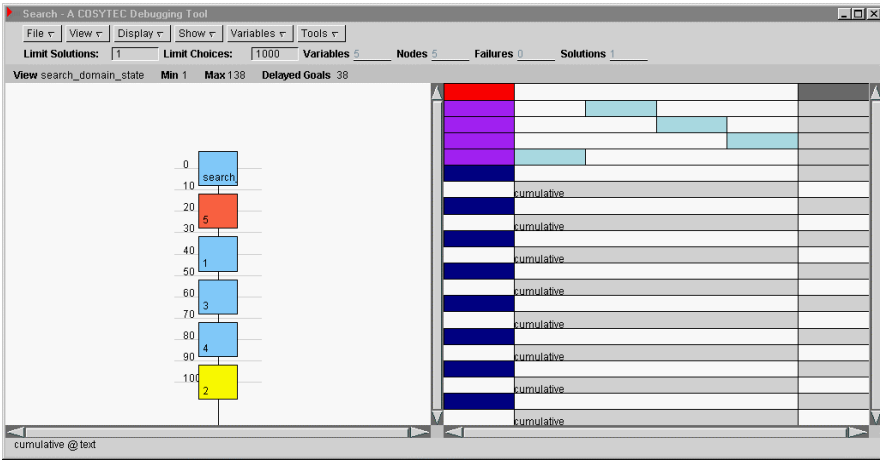


Fig. 13.27. Layout second phase: Propagation view

code responsible for this behaviour was developed a long time ago by other programmers, who do no longer maintain this application. Identifying this anomaly would have been difficult with standard debugging techniques.

Hence, with the search-tree tool analysis the constraint generation anomalies were fixed. Specifically, even if there were no modelling errors, we

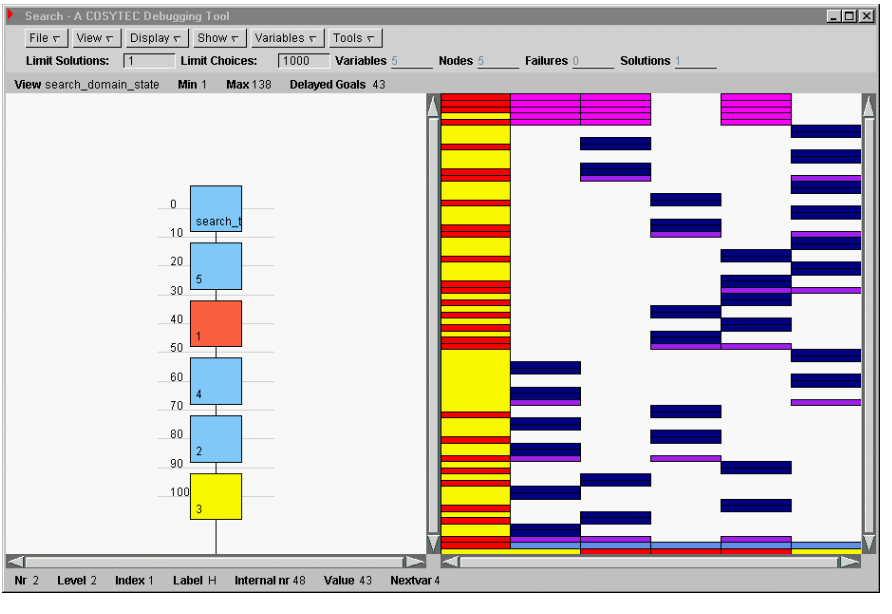


Fig. 13.28. Layout second phase: Incidence Matrix view

modified the generation of the element constraints to speed up their definition. The propagation and propagation events analysis was quite complex, but it allowed us to understand the actual CHIP behaviour when waking the constraints at a detailed level.

13.4 Analysis and Possible Improvements

All programmers started using the debugging tools on academic examples and course exercises before experimenting with existing and new industrial applications. The different users focused on different parts of the debugging tools: ICON concentrated more on the CHIP graphical propagation views, OM Partners concentrated more on the CHIP global constraint visualisers.

13.4.1 Need for a Debugging Methodology

For the tool developers the results obtained with industrial applications are certainly most interesting. However, a lot of time had to be spent learning how to interpret the information shown by the visualisation tools. Although it is rather easy to get the graphical debugging tools working, the users strongly recommend the development of a *graphical debugging methodology*. This debugging methodology could help the end user to exploit the visual information in the best way: which conclusions can be deduced immediately from the search-tree geometry and/or domain behaviour? What views are most interesting when going into more detail? Which functionality still has to be discovered?

13.4.2 Program Improvement Due to the Use of Debugging Tools

Before starting the assessment phase, it had been planned to evaluate the debugging tools by using statistics on the number of errors, the developing time, etc. when programming with/without debugging tools. For several reasons this was not feasible: First, given the time and resource limits of the project, industrial partners could not afford to train people or to develop new applications without using the best and most advanced tools. Next, since the graphical debugging tools only can be used when a program already runs, the debugging support for a large part of the program development has not changed. We still have:

- The typical syntactic and semantic errors. In general, they can be found using classical debugging techniques like text tracing. Here, the assertion tools could give assistance (see chapter 2).
- The errors due to data inconsistency. These inconsistencies often cause the violation of constraints and the immediate failure of the CP program (the famous "no" answer). The detection of these data inconsistencies is very time consuming and can in general not be done by simple text tracing.

Consequently, objective before/after comparisons were only possible when using the debugging tools in existing applications and looking to areas where the graphical debugging tools provide assistance, for example:

– the behaviour and efficiency of the labelling/optimisation routine,
– the efficiency of the constraint propagation,
– the effect of adding redundant constraints to the model.

All users have found that *all existing industrial applications for which the debugging tools have been used have been improved* (see detailed application descriptions of OM Partners, ICON and PrologIA in [13.8]). In most cases the labelling routine and/or propagation have been improved and result in a performance amelioration and/or better optimisation. However, the number of treated applications is not sufficient yet for detailed statistics.

13.4.3 General Conclusions on the CHIP Graphical Debugging Tools

The graphical representation of CLP problems through the search-tree tool and global visualisers clearly is what a user needs to understand and explore different search strategies. They encourage the user to optimise the search strategy and to go farther than the first feasible solution. Although most real-life problems still have to be scaled when using the debugging tools, the number of variables and constraints that can be used is big enough to test valuable industrial cases. As a result, new applications are written in such a way that the program can be started with/without debugging tools. The expert users are convinced that using the debugging tools will make the maintenance of industrial applications easier and less time consuming.

search-tree Tool. The overall view (shape of the search-tree) in the search-tree tool immediately gives an intuitive picture of how well the search strategy works (what is the degree of propagation). Afterwards, the user can inspect the execution in more detail, by clicking on the nodes of interest (comparing different nodes) and investigating the different views. The search-tree tool allows complete navigation through the tree, not just between adjacent nodes. Novice users reported that the constraint paradigm became immediately clear, expert users realized that a lot of difficult text tracing work, trying to understand what was happening, will be avoided in the future. For all users, the domain state and update views in the search-tree tool are quite easy to understand but working out the propagation view requires some more effort. The propagation event view is even less accessible for an end-user, but may no doubt provide essential information to the implementers of the system. Some suggestions for an easier and better search node annotation are given below. Keeping track of the search node number requires minimal effort when using CHIP objects (in that case the number is just an extra field in the object structure); it is not necessary to extend or build data structures to link the numbers with the variables. However, especially when dealing with

large problems, the user should have the possibility to define an application specific node name.

Global Constraint Visualisers. The global constraint visualisers offer graphical representations that are tailored to the specific problem at hand (to a specific use of the constraint). The fact that they can be used in combination with the search-tree tool results in a powerful tool set. Adapting the program to use the global visualisers was straightforward, as it only involves adding a simple wrapper around the global constraints. Combining the global constraint visualisers with the search-tree tool has the advantage that the visualisers are automatically updated when navigating through the search-tree. However, it also requires that all variables determining items (e.g. rectangles) in the visualisers are included as search-tree nodes. These extra search nodes complicate the views in the search tool, mixing up the relevant information with irrelevant details. Standalone use of the visualisers may provide more information than in combination with the search tree tool, e.g. if the predicate `min_max` is involved. The reason is that the search-tree tool currently does not remember constraints introduced during the search. The graphical user interfaces (GUI) of the global visualisers often come very close to dedicated application GUI. Although it is not the intention to compete with specialised GUI, it should be possible to extract more user defined information from the global visualisers. Suggestions for improvements can be found below.

13.4.4 Suggestions for Extensions to the CHIP Debugging Tools

User Definable Annotation. In the current tools, the annotation of the search nodes and items in the visualisers is predefined. Especially when dealing with large problems, this information is quite cryptic for the user. It is difficult to relate the annotation with the problem concepts (e.g. tasks to be scheduled). Moreover, there is no uniform notation (numbering) of variables/objects over the different visualisers and search-tree views. Hence, clicking on some item (rectangle) in a particular visualiser highlights items with corresponding numbers in other visualisers, but these do not necessarily correspond to the same task. A possible extension is to allow user-defined annotation. In this way, the user can specify information that is relevant for the specific problem at hand:

- In the search-tree tool, the `search_node` predicate could be extended such that the user can specify the node label, e.g.

  ```
  search_node(X, N, 'task3-product1', indomain(X))
  ```

  An extra option "Show User Label" in the search tool could then annotate the search node with this label.
- The visualise wrappers could also be extended to allow user-defined annotation of the items in the visualisers. E.g. an extra argument could be the list of labels corresponding to the list of rectangles in the `diffn`, or the list of variables (starts, duration, ends,..) in the `cumulative`.

A further extension would be user-defined colouring of search nodes or visualiser items based on some characteristic of the underlying problem concept (e.g. based on task type). Based on these annotations the user could identify the search-tree nodes that are interesting according to a certain criterion, and consequently decide which part of the tree to observe.

Tree Information.

- It would be interesting if the user could easily identify on the tree some areas sharing common characteristics, for instance big domain reductions, isomorphic sub trees, unaccounted problem symmetries, etc. The availability of statistics could be quite helpful to let the user compare the quality of different strategies.
- Another possible improvement is to highlight/colour the nodes that (do not) satisfy specific conditions. For instance, during performance debugging, the user should be able to quickly identify `min_max` failure nodes (i.e., solutions worse than a previous one) in order to refine the heuristic. With the present tool, the user must select any failure node and find out if the cause of failure is interesting.
- No information is provided about *shallow backtracking*, i.e. backtracking within a search-tree node. Such information is also relevant when comparing different labelling strategies and their degree of propagation. A proposal is to display the number of backtracks within each search-tree node.
- Constraints introduced during the search are currently not remembered by the search-tree tool (e.g. the extra constraint introduced by `min_max` after a first solution has been found). This results in a wrong image in the domain state view (the domain may actually be smaller than shown). The correct domain can only be seen in the indication of the domains in the search nodes.

View Manipulation Utilities. The user would be helped during both correctness and performance debugging if the tool gave some assistance in the search-tree exploration. Some possible improvements could be the following ones:

- Collapsing the nodes that represent variables becoming ground because of propagation. Actually, a typical event is the reduction of variable domain to a single value, due only to constraint propagation following some assignment. This fact is translated on the search-tree into node chains where there is no additional domain reduction. The graphical representation of this property would avoid that the user analyses these nodes, which do not carry any extra information.
- Defining variable order in search-tree views. It would be helpful to let the user change the order of variables in views, in order to have a better visual representation for certain problems without tweaking with the `search_number` adornment. More generally, it would be advisable to simplify the search-tree tool wrapper, in order to avoid the reordering of traced

variables inside the code (see above). Another solution could be to allow the user to dynamically change the visualisation and ordering of the elements inside the tool with mouse actions, for instance by means of drag-and-drop of rows and columns, or with pop-up menus.

New Views.

- User definable representations (based on existing view dimensions) The 2D representation of domains is suitable. As a possible extension, we suggest a more general framework where the user defines his/her own representations. The idea is that if the tool provides some "dimensions" (e.g., list of the variables, list of the domains, constraints, propagation, etc.), the user could select as horizontal or vertical axis the preferred items, possibly selecting subsets or organising them in parallel layers, in order to represent/visualise also multidimensional problems.
- A utility to compare graphical views pertaining to different nodes. Very often during the analysis it is important to understand the ongoing domain reduction. In the current tools, the user selects in turn two different nodes in order to depict the changes. It would be better to have a "diff" utility that represents graphically the differences in a view between two nodes (e.g., a "Diff Domain State" view).

Extension of the Global Constraint Visualisers.

- Standalone visualisers: breakpoint with user interaction. During standalone use of the visualisers, the user should have the possibility to insert a breakpoint upon which some action can be taken. E.g. it would be nice if the user could ask for additional information, such as the domain of a variable, etc. Currently, one has to decide in advance which information has to be displayed. No interaction / extension is possible once program execution has started.
- 3D diffn_visualizer. A visualiser of 3D problems could help in case of bin packing problems.
- Better scrolling and zooming functionality. The current zoom facilities do not change the precision of the scales when zooming to a small part of the original range.

13.5 Conclusions

In this chapter we have described some practical experience with the use of the CHIP visualisation tools. We have presented how they can be used to detect and overcome typical problems of performance debugging. These rather general rules are a first step to a more refined methodology of performance debugging with visualisation tools. In a second part, we have presented a number of industrial applications that were checked during the assessment phase of the DiSCiPl project. These applications had been developed before

the tools were available, but in each instance some improvement of the program was discovered by using the different tools.
The current state of the visualisation tools is just a beginning. In the last section of the chapter, we present some comments on the existing tools and ideas for their improvements. Clearly, the existing tools are already very useful for developing industrial constraint applications, but more work is still required to make them more accessible for programmers and to allow an easier interpretation of results.

References

13.1 A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems. Journal of Mathematical and Computer Modeling, Vol. 17, No. 7, pages 57-73, Pergamon Press, 1993.
13.2 N. Beldiceanu, E. Bourreau, P. Chan, and D. Rivreau. Partial Search Strategy in CHIP. 2nd International Conference on Meta-heuristics, Sophia-Antipolis, France, July 1997.
13.3 N. Beldiceanu, E. Bourreau, D. Rivreau, and H. Simonis. Solving Resource-Constrained Project Scheduling Problems with CHIP. Fifth International Workshop on Project Management and Scheduling, Poznan, Poland, April 1996.
13.4 N. Beldiceanu, E. Bourreau, and H. Simonis. A Note on Perfect Square Placement. COSYTEC Technical Report, January, 1999.
13.5 N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.
13.6 E. Bourreau. Traitement de Contraintes sur les graphes en programmation par contraintes. PhD thesis, L.I.P.N.,Université Paris 13, March 1999.
13.7 Y. Caseau and F. Laburthe. Cumulative Scheduling with Task Intervals. Proceedings of the Joint International Conference and Symposium on Logic Programming, M. Maher (Ed), The MIT Press, 1996.
13.8 T. Cornelissens. General report on assessment of the tools. DiSCiPl deliverable D.WP1.3.M1.3, April 30, 1999.
13.9 I.P. Gent and T. Walsh. CSPLIB: A Benchmark Library for Constraints. In J. Jaffar (Ed), Principles and Practice of Constraint Programming, CP99, Alexandria, VA, pages 480–481, October 1999.
13.10 M. Fabris et al. CP Debugging Needs and Tools. In Proc. of the 3rd. International Workshop on Automated Debugging-AADEBUG'97, Pages 103-122, Linköping, Sweden, May 1997.
13.11 M. Gardner. Scientific American, April 1975.
13.12 ROSEAUX. Exercices et Problèmes Résolus de Recherche Opérationnelle. Tome 3, Paris, p.279–282, 1983.
13.13 H. Simonis and A. Aggoun. Search Tree Visualization. COSYTEC Technical Report, DiSCiPl deliverable D.WP3.1.M1.1-2, September 1997.
13.14 H. Simonis. Visualization in Constraint Logic Programming. Invited Tutorial. PACLP 99, London, UK, April 1999.
13.15 P. Van Hentenryck. Constraint Satisfaction in Logic Programming. The MIT Press, Boston, Ma, 1989.

Author Index

Subject Index