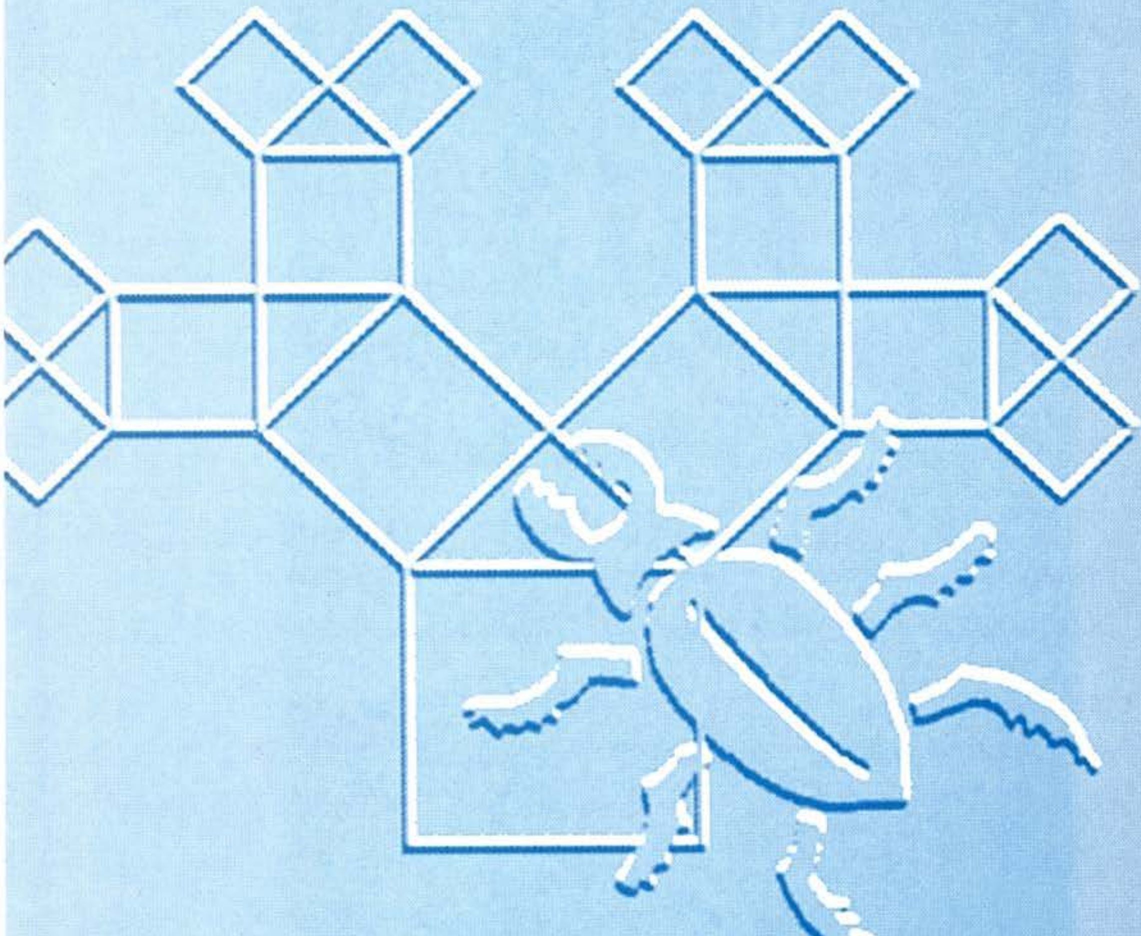
A Framework for Debugging Heterogeneous Applications

Pieter Olivier



A Framework for Debugging Heterogeneous Applications

On Bugs and Trees

INSTITUUT VOOR PROGRAMMATUURKUNDE EN ALGORITMIEK

ipa

The work in this thesis has been carried out
under the auspices of the research school IPA
(Institute for Programming research and Algorithmics).

A Framework for Debugging Heterogeneous Applications

On Bugs and Trees

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.dr J. J. M. Fransse
ten overstaan van
een door het college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 5 december 2000, te 10.00 uur

door
Pieter Antonius Olivier
geboren te Monnickendam

Promotor: prof.dr P. Klint
Co-promotor: dr M.G.J. van den Brand

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Olivier, P.A.

A Framework for Debugging Heterogeneous Applications. On Bugs and Trees / Pieter A. Olivier.
- Amsterdam : Universiteit van Amsterdam.
- Proefschrift Universiteit van Amsterdam.
- Met illustraties, register en literatuur opgave.
- Met samenvatting in het Nederlands.
ISBN 90-9014210-X
NUGI 855
Trefwoorden: debugging; gedistribueerd / boomstructuren

Printed by Universal Press, Veenendaal.

Acknowledgements

After grinding on this work for six years, both at the UvA and at the CWI, I finally arrived at the most pleasant and at the same time most difficult part of this thesis: thanking the people who have made a significant contribution to my work and life during this period.

First and foremost I would like to thank Paul Klint, who hired me even after I showed up an hour late for our first appointment :-). Since then he has proved to be an inspiring mentor and coach who never seemed to get tired of listening and contributing to all the strange ideas and plans I've discussed with him over the years. Paul, thanks for being my promotor and thanks for the incredibly interesting working environment you have created. I will be forever grateful for the freedom you have given me to pursue my own (research) ideas.

Another person I owe a lot to is my co-promotor and friend Mark van den Brand. Mark has been the driving force behind the development of the ASF+SDF Meta-Environment from the time that I starting working on it. His tireless efforts and enthusiasm have been a great motivation that have helped me to overcome many obstacles, both when writing articles and when writing software.

Hayco de Jong has helped me to hone my programming skills in several areas, not in the least by teaching me that taking a break now and then can increase the quality of your work significantly. I've tremendously enjoyed the many hours we have spent 'pair-programming' together.

I would like to thank the members of the reading committee, Prof.dr. Henry Bal, Prof.dr. Jan Bergstra, Dr. Dave Hanson, Dr. Anne Kaldeway, and Prof.dr. Martin Kersten, for their useful comments that helped to improve the overall quality of this thesis.

My colleagues have contributed a lot in making the daily working environment a pleasant one. I would especially like to thank
Arie van Deursen for his funny PEM interruptions,
Bob Diertens for his perl knowledge,
Jan Friso Groote for always providing new performance challenges for our software,
Jan Heering for his incredibly broad expertise,
Merijn de Jonge for his system programming skills,
Jan Kort for his dedication to functional languages,
Tobias Kuipers for his sceptical sense of humor,
Ralf Lämmel for explaining the notion of *Kamerad Betrüger*,
Leon Moonen for sharing some of his juggling and climbing skills,

Alban Ponse for providing answers to my process algebra questions,
Piet Rodenburg for his outdoor course on module algebra,
Jeroen Scheerder for his operating system knowledge,
Chris Verhoef for his unique sense of humor and weird taste for music,
Jurgen Vinju for his sunny disposition and wild imagination,
Eelco Visser for our interesting discussions and for his beautiful but unreadable specifications,
Joost Visser for his critical remarks at the right time.

My parents always supported my urge to want to know stuff, and they always kept an open mind as to my decisions what education to follow. Their only requirement was that I did my best to finish whatever I started. I hope they feel this thesis satisfies this requirement.

Last but certainly not least I would like to thank Maud Weddepohl for her love and kindness during these stressful times. She has provided the greatest support of all, not in the least because she is the only one in this world who understands my emotions as well as my research.

Contents

1
Introduction

1.1 Motivation

There is a rapidly growing interest in tools that help with the understanding, analysis, and debugging of programs. Fueled by the *year 2000* and *euro conversion* problems, the awareness has grown that software systems are not static entities that are created once and used unchanged until they are replaced by new versions. Software needs to be changed and extended in order to remain effective in an ever changing environment.

An aspect of this *software maintenance* that is often underestimated is the fact that not only the software evolves over time, but also the knowledge about the software. The pure complexity of most software systems makes it very hard to document every aspect of its design and implementation. Keeping this documentation synchronized with the ever evolving software is even more challenging. Personnel changes and the human defect to forget things as time passes make sure that part of the knowledge about the software will inevitably disappear. When a lot of the knowledge of the intrinsic workings of a software system has been lost, the system is called a *legacy* system.

There are two ways in which this lost knowledge can be retrieved: reading the source code of the software and by observing the runtime behavior of the software. These two methods are commonly referred to as *static* and *dynamic* analysis of software. The act of gathering information about existing programs by studying the source code and/or runtime behavior is called *reverse engineering*.

A powerful weapon in the battle against legacy systems is the use of languages and language constructs that offer abstractions closely related to the problem domain. By using these *domain specific languages*, the distance between the actual source code of the software and the documentation describing the intended operation of the system at the problem domain level is decreased considerably. This makes it easier to bridge this gap when there is a need to understand the software in order to make changes or even replace it with new software. Many problems related to the maintenance of software are directly related to the size and complexity of the source code. The use of domain specific languages can be of great help in this area because problem solutions can be expressed by using domain specific constructs instead of the more general (and often more low-level) constructs found in general purpose languages.

It is important to realize that with the use of domain specific languages, part of the software development and maintenance effort shifts from the application program level to the language level. The main reason why general purpose languages have been (and will be) so successful is that tools for them are readily available and can be bought at a fraction of the cost of developing them in-house. In order for domain specific languages to be successful, the amount of resources needed to design, develop, and maintain tools for these new languages must be lower than the amount of resources that can be saved by using them.

Two effective techniques for reducing the costs of building new tools for domain specific languages are the use of general parameterized tools and the automatic generation of tools based on formal language definitions. The former technique is well known for its use in relatively simple tools like syntax directed editors and pretty printers. In part I of this thesis, we will present a more complex parameterized tool: a language independent source level debugger.

The latter technique is less well known, and has been the main focus of the GIPE (Generation of Interactive Programming Environments) research project. One of the key accomplishments of this project has been the development of the algebraic specification language ASF+SDF and the ASF+SDF Meta-Environment which is an interactive language development environment that can automatically generate entire interactive programming environments based on a formal language definition in ASF+SDF.

Part II of this thesis shows how ASF+SDF specifications can be executed using both interpretation and compilation techniques. We will also show how the techniques from Part I can be used to debug ASF+SDF specifications themselves, and how they can be used to add debugging support to languages developed using ASF+SDF.

1.1.1 Source Level Debugging

In Part I of this thesis we will focus on a particular kind of language specific tools that is used for the analysis of the runtime behavior of software, the *source level debugger*. The primary goal of source level debuggers is to visualize different aspects of the execution of a software system at the source code level. By providing control over the execution speed of the program, and by providing visual feedback on the evolving internal state of the program, a source level debugger helps the user to understand the internal workings of the software being studied.

Source level debuggers have always been seen as inherently language specific tools, and often even as language implementation specific. It would be very expensive to develop a new debugger for every domain specific language. Consequently, most domain specific languages in existence today have no debugging support whatsoever. The most predominant way of gathering information about the execution of a program is still the error-prone and time consuming insertion of print statements.

We will show that the only thing about source level debugging that is really language (implementation) dependent, is the way in which primitive debugging events are generated. The bulk of the debugger functionality, including all of the user interface and the way in which the primitive debugging events are visualized, is completely language independent. We will present a debugging framework and implementation that can be parameterized with such primitive event gathering mechanisms making it cost effective to build debuggers for both domain specific languages and general purpose languages.

1.1.2 Execution of ASF+SDF Specifications

ASF+SDF [BHK89, DHK96] is a modular algebraic specification formalism for describing the syntax and semantics of (programming) languages. SDF [HHKR92] (Syntax Definition Formalism) allows the definition of the concrete and abstract syntax of a language and is comparable to (E)BNF. ASF (Algebraic Specification Formalism) allows the definition of the semantics in terms of equations, which are interpreted as rewrite rules. The development of ASF+SDF specifications is supported by an integrated programming environment, the ASF+SDF Meta-Environment [Kli93].

ASF+SDF can be seen as a domain specific language itself, targetted at developing programming languages. This is why the ASF+SDF interactive programming environment is called the ASF+SDF *Meta-Environment*. The purpose of this ASF+SDF Meta-Environment is not only to support specification and development of new languages, but also to support specification and generation of *language specific tools*. In order for these tools to be used successfully in a wide range of applications, they need to execute efficiently both in terms of *time* and *space* requirements.

Part II of this thesis focuses on a number of aspects related to the efficient execution of ASF+SDF specifications. We will present a compiler (which itself is written in ASF+SDF) that compiles ASF+SDF specifications to C. The most important datatype used in the resulting C code is called the *ATerm datatype*, and it plays a central role in the efficient execution of ASF+SDF specifications. We will show that by basing the design and implementation of this datatype on *maximal subterm sharing* and *efficient garbage collection*, we can generate tools that can process huge amounts of data. This opens up wider application domains than would otherwise be possible to address.

Specifications can become quite large, making it impractical to recompile them whenever they are only slightly modified. Especially during the development of specifications, it is important to have a short turnaround time when trying out new ideas or fixing bugs. To make this possible we also developed an *interpreter* to execute ASF+SDF specifications. The interpreter not only makes it possible to test changes to a specification more quickly, it also makes it possible to experiment more easily with changes to the execution model and semantics of ASF+SDF itself. Changing the compiler to experiment with these kind of modifications would be much too cumbersome and time consuming. One of the key design considerations for the interpreter has been that its architecture should be open for experimentation and that it should be extensible, even at the cost of raw execution speed.

1.1.3 Debugging of ASF+SDF Specifications

As ASF+SDF specifications become larger, understanding them and keeping track of what is going on at execution time can be very difficult. Just as with other domain specific languages, developers of ASF+SDF specifications need a good debugger to gain understanding of their specifications and to find bugs. We will show how the debugger framework discussed in Part I can be used to create a debugger for both compiled ASF+SDF specifications and interpreted ASF+SDF specifications.

When developing a new domain specific language using ASF+SDF, often one of the tools you would like to create is a debugger for debugging programs written in this new language. In a case study we will show how a compiler specified in ASF+SDF can be adapted to instrument the generated code with debugging support. Using this technique, our debugging framework can be used to debug programs *generated* by the tools generated from ASF+SDF specifications!

1.1.4 Research Questions

The material in this thesis has been structured around three central research questions. These questions were raised by my work on debugging and execution of algebraic specifications, and have in a way guided my research in these areas. At the end of this thesis we hope to provide the reader with answers to these questions. In answering these questions we will undoubtedly raise some new and interesting (research) questions.

The first question is related to the notion of *generic debugging*, i.e. debugging by generalizing the semantics of a programming language to a level that covers a whole set of languages.

Research Question 1: *Is it possible to develop generic debugging technology that can be used to significantly reduce the cost of developing debuggers for new languages?*

The second question is related to the use of a technique called *maximal term sharing* in the execution of algebraic specifications in an industrial setting.

Research Question 2: *Can maximal term sharing be used to increase both the time and space efficiency of executable algebraic specifications?*

The last question is about the applicability of generic debugging to ASF+SDF.

Research Question 3: *Can generic debugging technology be used in the* ASF+SDF *Meta-Environment at the following three levels: Debugging of the* ASF+SDF *Meta-Environment itself, debugging of* ASF+SDF *specifications, and debugging of programs written in languages specified in* ASF+SDF*?*

1.2 Overview of this Thesis

In the first part of this thesis we present the generic debugging framework we developed. In Chapter 2 we start by introducing the TOOLBUS coordination architecture around which the framework is built. Chapter 3 presents a case study in which we gained experience with using the TOOLBUS to implement a medium sized distributed system. Using this case study, we also gained insight in the feasibility of some of our ideas about generic debugging. In Chapter 4 we discuss our generic debugging framework, and present the TIDE debugger, which is an implementation of this framework.

In the second part of this thesis we will discuss the compilation and debugging of algebraic specifications. We will start by providing some context for this work, in the form of an overview of the new ASF+SDF Meta-Environment we are currently developing (Chapter 5), which is largely based on the ideas and techniques discussed in this thesis. In Chapter 6 we discuss the design and implementation of the lowest layer of our work: the ATerm library. Chapter 7 shows how ASF+SDF specifications can be compiled to C code, and how we use the ATerm library as a foundation on which the runtime support for the compiled code is based. In Chapter 8 we make the connection between parts I and II of this thesis by showing how TIDE can be used to add debugging support to both the ASF+SDF compiler and interpreter, making it possible to debug ASF+SDF specifications. We conclude Chapter 8 with a case study that shows how TIDE can also be used to generate debugging support using a compiler specified in ASF+SDF.

1.3 Main Contributions

The work described in this thesis contributes to two areas in the field of computer science. Not surprisingly, this thesis is split into two parts centered around these areas.

The first area is that of debugging heterogeneous distributed applications. The idea that such applications can be debugged using a single debugger (or at least a single debugger front-end) has to our knowledge not been pursued before. We show that this kind of integration is quite feasible by building on today's state-of-the-art low level debugger implementations.

The second area is that of space and time efficient execution of algebraic specifications. Most work in this area up to now has focussed primarily on time efficient execution. Because of our interest in generating industrial strength applications like analysis and transformation tools for large Cobol systems, space efficiency is equally important. By using maximal subterm sharing, a technique that is also referred to as 'hash consing' in the (Lisp) literature, we succeeded in substantially reducing memory requirements, often by one or more orders of magnitude. We do so without seriously compromising execution speed. In fact, we will show that in some cases the reduced memory requirements actually result in a gain in execution speed. Based on this evidence, we claim that our implementation is one of the first truly successful applications of maximal subterm sharing.

1.4 Related Work

This thesis contains several sections discussing related work. In Section 2.8, we compare the TOOLBUS to some of the mainstream software interconnection architectures like DCOM and CORBA. We will argue that the TOOLBUS should not be considered a competitor to these standards, but rather a unifying architecture that offers a higher level of abstraction than these low level 'wiring standards'.

In Section 4.6 we relate our work on portable and multi-lingual debuggers with other work in that area, and especially with the work on debuggers like **ldb** [RH92] and **cdb** [HR96, Han99b], which have been developed to supply portable debugging support for the retargetable C compiler **lcc** [FH95].

In Section 6.6.1 we give an overview of some of the work related to intermediate representations of tree-like data structures like ATerms, that are typically used in compiler frameworks. We also relate our use of ATerms for the execution of algebraic specifications to improve the space and time efficiency with the use of hash consing in LISP [All78] and experiments with sharing in SML [AG93].

1.5 Origins of the Chapters

Many of the chapters in this thesis are revised versions of publications that have previously appeared elsewhere.

- **Chapter 3: A Simulator Framework for Embedded Systems**
  This chapter is based on an article that appeared in COORD'96 (Coordination Languages and Models) [Oli96b].
- **Chapter 4: Debugging Heterogeneous Distributed Applications**
  Based on the article *Debugging Distributed Applications using a Coordination Architecture* [Oli97], which appeared in COORD'97 (Coordination Languages and Models).
- **Chapter 5: The ASF+SDF Meta-Environment**
  Based on joint work with M.G.J. van den Brand, T. Kuipers and L. Moonen, appeared as *Implementation of a Prototype for the New* ASF+SDF *Meta-Environment* [BKMO97] in the conference proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.
- **Chapter 6: The ATerm Library**
  Based on an article that appeared in Software, Practice and Experience [BJKO00]. This is joint work with M.G.J. van den Brand, P. Klint and H.A. de Jong.
- **Chapter 7: Compiling ASF+SDF Specifications**
  Based on joint work with M.G.J. van den Brand and P. Klint, which appeared as *Compiling* ASF+SDF *specifications* [BKO99] in CC'99 (Compiler Construction).

Note that we have tried to eliminate most (but not all) sources of redundancy between these chapters in such a way that all chapters are still readable separately. In some cases the same subject is still discussed several times from a different point of view, in which case we feel that it would only decrease the readability of this thesis if we tried to eliminate or cluster discussions.

The most striking example of this is the presentation of maximal sharing in the ATerm library. This subject is first touched upon in Chapter 5, explained thoroughly in Chapter 6, and discussed again in Chapter 7.

In all three cases the same subject is discussed, but in each case a different point of view is taken, and different aspects of maximal sharing are highlighted.

1.6 About the Implementations

The amount of implementation work presented in this thesis is substantial. I would like to stress that most of this work is the result of the collective implementation effort of quite a few researchers. In this section, I will give an overview of the most important software components presented in this thesis, introduce the authors of these components, and explain what my contribution has been.

I will also give an estimate of the size of the different components using lines of code as a metric. Although such a metric is not very accurate, it does give an impression of the implementation effort invested in these tools.

The TOOLBUS*(Chapter 2, [BK98])* The original TOOLBUS script interpreter was written in ASF+SDF. Paul Klint implemented it in C, resulting in an application that consists of about 16,000 lines of code.

My main contributions to the TOOLBUS implementation effort are finding and fixing some of the bugs from the early versions, and the implemention of some of the adapters that make it possible to connect tools written in different languages to the TOOLBUS (the Tcl/Tk adapter, the Python adapter, the SWI Prolog adapter, an early ASF+SDF adapter, the epic adapter, and the Java adapter, with a total of about 12,000 lines of C and Java code).

The Simulator Framework (Chapter 3, [Oli96b]) All implementation work done on this project has been my responsibility. The final implementation includes a development system consisting of an lcc backend, assembler, linker and C library, and a simulator consisting of a virtual machine and several user interface components. The complete system is implemented using about 43,000 lines of code in a wide range of programming languages.

The TIDE implementation (Chapter 4, [Oli97]) This project has also been completely in my hands, except for some work on expression animation that was done by Hayco de Jong in the context of his masters thesis [dJ99]. Although still being actively extended, the current version consists of around 15,000 lines of code, primarily Java.

The ASF+SDF *Meta-Environment implementation (Chapter 5, [BKMO97])* The implementation effort on this project spans more than a decade and has been done by numerous researchers. It is therefore not possible to name everyone who has contributed in the course of the project. Among the people who are currently working on this project are: Mark van den Brand (compiler, interpreter, module database, and parser generator), Merijn de Jonge (configuration management), Tobias Kuipers (editors), Leon Moonen (user interface), and Jeroen Scheerder (parser). I have mainly been involved in the implementation of the compiler and interpreter.

The current implementation consists of about 24,000 lines of C code, about 2,500 lines of Java code, and over 19,000 lines of ASF+SDF.

The ATerm library (Chapter 6, [BJKO00]) The ATerm implementation has been a joint effort of the author of this thesis with Hayco de Jong. Together we developed this heavily optimized library that consists of roughly 12,000 lines of C code, and 5,000 lines of Java code.

The ASF+SDF *compiler (Chapter 7, [BKO99])* The implementation of the ASF+SDF compiler has already been briefly discussed in the context of the ASF+SDF Meta-Environment implementation. Mark van den Brand was the primary author of the actual compiler, which is specified completely in ASF+SDF. I was responsible for developing the runtime system of the compiler.

The compiler consists of about 9,000 lines of ASF+SDF. The runtime system is based on the ATerm library discussed earlier, combined with about 1,000 lines of glue and interface code.

Part I

Generic Debugging Technology

2

TOOLBUS

The TOOLBUS [BK98] is a software coordination architecture that utilizes a scripting language based on process algebra [BW90] to describe the communication between software tools. A TOOLBUS *script* describes a number of processes that can communicate with each other and with *tools* existing outside the TOOLBUS. In the current TOOLBUS implementation, every tool is implemented as a single operating system process. A language dependent adapter that translates between the internal TOOLBUS data format and the data format used by the individual tools makes it possible to write every tool in the language best suited for the task(s) it has to perform.

Most of the work in this thesis is based on the TOOLBUS software coordination architecture. In some sense it is the *leitmotif* for our work, so it seems only appropriate that we start this first part by introducing the TOOLBUS.

One of the most striking developments in the software industry over the last 10 years is the shift towards *component based software development*. Modern software systems are often designed using a number of loosely coupled components. These components are then combined using a *component* or *coordination* architecture. Typical advantages of a component based approach are increased software reuse and easier maintenance.

Some of the well known component architectures used in industry today include OMG's `CORBA` [Cor99], Microsoft's `DCOM` and SUN's `JavaBeans` [Ham99]. However, these systems are fairly low level, and can also be described as *wiring standards*. More high level coordination architectures that are actually used in an industrial setting are very hard to find, but the Java based `InfoBus` from SUN comes close. A good discussion on the use of component technology in general and `CORBA` and `DCOM` in particular can be found in [Szy97].

The TOOLBUS architecture is characterized by its formal basis in the form of process algebra [BW90], and the use of a generic data exchange format makes this architecture truly language independent.

A TOOLBUS *script* describes a number of processes inside the TOOLBUS that can communicate with each other and with *tools* existing outside the TOOLBUS (Figure 2.1).

It is important to note that TOOLBUS tools can *only* communicate with processes in the TOOLBUS. They are not allowed to communicate directly with each other without interven-

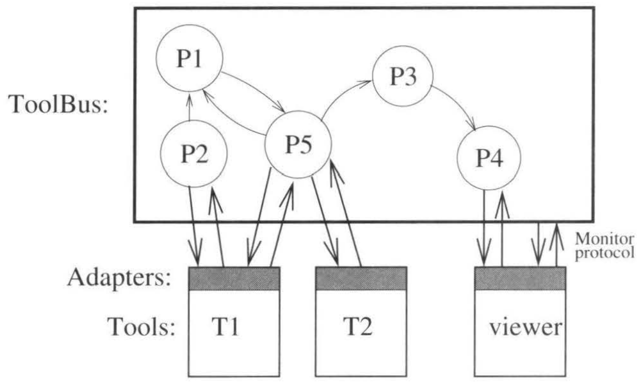


Figure 2.1: The TOOLBUS software application architecture

tion of the TOOLBUS. This ensures that the TOOLBUS script has complete control over the communication between tools.

Each tool is connected to the TOOLBUS using an *adapter*. An adapter is some interface code that is responsible for the connection with the TOOLBUS, and converts between "native" data formats used inside a tool and the common data exchange format used by the TOOLBUS. TOOLBUS adapters are language specific. This means each language (implementation) needs its own adapter to connect to the TOOLBUS. At this point we have about a dozen adapters for a wide range of languages varying from Tcl/Tk to COBOL.

TOOLBUS processes are described as process expressions which are built using primitive TOOLBUS *actions* and process composition operators. The following sections give an overview of the most important TOOLBUS primitives and operators. A complete overview can be found in Appendices A, B, and C.

2.1 Communication between TOOLBUS Processes

There are two mechanisms available for processes in the TOOLBUS to communicate with each other, message passing and selective broadcasting. A process can synchronously send a *message* using the `snd-msg` primitive which must be received by another process using the `rec-msg` primitive. Both of these primitives take a variable number of arguments. Data is transfered between sending and receiving process using *matching*. The argument terms of the `snd-msg` and `rec-msg` primitives are matched against each other. These argument terms can contain result variables, indicated by a trailing question mark (`?`). When such a variable matches with a subterm, that subterm is assigned to the variable. For example, if communication occurs between:

```
snd-msg(text("Hello world!"))
```

and

```
rec-msg(text(T?))
```

The value `"Hello world!"` is assigned to `T`.

A process can send a *note* using `snd-note` to all processes that have subscribed, using `subscribe`, to that particular note type. The receiving processes read notes asynchronously using `rec-note`, at low priority. Transmitting notes amounts to *asynchronous selective broadcasting*. Data transfer is again accomplished using matching.

2.2 Communication between TOOLBUS and Tools

A TOOLBUS process can initiate communication with a tool by sending a message to a tool using `snd-do`, or `snd-eval` when an answer is expected. A process can receive the answer to a `snd-eval` request using the `rec-value` action.

A tool can initiate communication by sending an *event* to the TOOLBUS. A TOOLBUS process receives this event using the `rec-event` primitive and must acknowledge the event using the `snd-ack-event` primitive.

The execution and termination of the tools attached to the TOOLBUS as well as their connection/disconnection can be controlled explicitly. The `execute` action starts a new tool, while the `rec-connect` action waits for a tool to connect itself to the TOOLBUS. The `snd-disconnect` action actively disconnects a tool, while the `rec-disconnect` action waits until a tool disconnects itself from the TOOLBUS.

2.3 Process Composition

More complex processes can be created using process composition operators for *choice* (+ operator), *sequential composition* (`.` operator), *parallel composition* (`||` operator), *iteration* (binary `*` operator) and *guarded (conditional) execution* (the `if-then-fi` operator). The test part of the `if-then-fi` uses *expressions* as discussed in the next section. The *process creation* primitive `create` can be used to create new process instances.

Process algebra semantics of the + operator (and consequently the `||` operator because its semantics are expressed in terms of + and `.`) demand a non-deterministic choice between the alternative process expressions. In the TOOLBUS this is modeled by random selection of alternatives when both alternatives are valid.

2.4 Types and Variables

The TOOLBUS uses a common datatype called *ATerm* to represent data. ATerm stands for *Annotated Term*. The annotation mechanism is not visible from within TOOLBUS scripts, and will be discussed in Chapter 6 in more detail. Only ATerms can be exchanged between tools and the TOOLBUS.

A limited set of operations is available from within TOOLBUS scripts to analyze and transform these terms.

All terms within the TOOLBUS are *typed*. The TOOLBUS defines a number of basic types for booleans, integers, reals, strings, and binary strings. Complex types can be formed using a list constructor or function application. The type `term` is a supertype of all other types.

The `let-in-endlet` construction makes it possible to declare variables. Assignment to variables is possible via the `:=` operator. On the right-hand side of this operator, *expressions* can be given using a number of prefix functions operating on values over the basic types. Typical functions include the (in)equality check (`equal` and `not-equal`), functions operating on booleans like `and`, `or` and `not`, and functions operating on lists like `first` and `next`. Appendix C contains a complete overview of all functions available in expressions.

2.4.1 A Producer/Consumer Example

In Figure 2.2 an example is presented that models the relationship between a producer and multiple consumers. The producer produces numbers that are printed by a pair of consumer processes. Note that this example uses some TOOLBUS primitives that have not been discussed yet, namely the `printf` action that can be used to write output to the screen, and the `delta` action that represents deadlock. Figure 2.2 also shows the output of the TOOLBUS script. `product` is a user-defined function symbol used to match corresponding `snd-msg` and `rec-msg` actions. Note that because the `delta` action is never executed, it can effectively be used in combination with the iteration operator to implement an endless loop.

2.5 A Calculator Example

In this example a TOOLBUS script is presented that connects two tools, a calculator tool that calculates expressions and a user interface tool that asks the user for an expression and presents its value as result.

This TOOLBUS script contains two processes, a `USER-INTERFACE` process (Figure 2.3) that handles user interface events and a `CALC` process (Figure 2.4) communicating with the calculator tool.

In addition to these processes, two tools are introduced by *tool declarations*. The string following `command =` will be executed as a command by the underlying operating system to create an instance of the tool. A tool declaration also introduces a new type, that can later be used to declare tool identifier variables of that type.

The `USER-INTERFACE` process (Figure 2.3) uses three variables. The first one, `UI`, is a tool identifier of type `ui`. The second variable `E` contains an expression to be calculated, the third variable `V` contains the calculated result.

The `USER-INTERFACE` process first starts the user interface tool. The variable `UI` is a

```
%% The producer produces consecutive numbered products.
process PRODUCER is
let
  N : int
in
  N := 1 .
  ( snd-msg(product(N)) .
    N := add(N,1)
  ) * delta
endlet

%% The consumer consumes any product it can get hold of.
process CONSUMER(ID : term) is
let
  N : int
in
  ( rec-msg(product(N?)) .
    printf("CONSUMER(%t) consumes product %d\n", ID, N)
  ) * delta
endlet

toolbus(PRODUCER, CONSUMER(1), CONSUMER(2))

------ Output ------
CONSUMER(1) consumes product 1
CONSUMER(2) consumes product 2
CONSUMER(2) consumes product 3
CONSUMER(1) consumes product 4
CONSUMER(2) consumes product 5
...
```

Figure 2.2: Producer/consumer example

result occurrence of `UI`, because it is followed by a question mark (`?`). When a variable is used as a result variable a value is assigned to it, in contrast with a *value occurrence* of a variable (without a following `?`), when the current value of the variable is substituted. In this case, the tool identifier for the new instance of the user interface tool (`ui`) is assigned to `UI`.

After starting the `ui` tool, the `USER-INTERFACE` process enters a loop waiting for `expr` events from the newly created user interface tool. Such an event is generated when the user enters an expression and wants to evaluate it. At this point the user interface tool generates an

```
tool ui is { command = "wish-adapter -script ui-calc.tcl" }

process USER-INTERFACE is
let
  UI : ui,
  E : str,
  V : term
in
  execute(ui, UI?) .
  ( rec-event(UI, expr(E?)) .
    snd-msg(calc, expr(E)) .
    rec-msg(calc, expr(E, V?)) .
    snd-ack-event(UI, expr(E, V))
  ) *
  rec-event(UI, quit) .
  snd-ack-event(UI, quit) .
  shutdown("Goodbye!")
endlet
```

Figure 2.3: The `USER-INTERFACE` process

`expr` event, for instance `expr("3+4")`. The expression `"3+4"` is assigned to the variable `E`, and sent to the `CALC` process for evaluation using the `snd-msg` action. The result is received in the `rec-msg` action and returned to the user interface tool using `snd-ack-event`.

The loop continues until the user interface tool generates a `quit` event, for instance when the user presses the `Quit` button.

Now we turn our attention to the `CALC` process (Figure 2.4). It starts the calculator tool and waits for calculation requests. It sends them to the calculator tool and sends the result back.

The last construct of every ToolBus script is the ToolBus configuration that starts a number of processes in parallel. In this case the processes `USER-INTERFACE` and `CALC` are created and execution begins:

```
toolbus(USER-INTERFACE,CALC)
```

```
tool calc is { command = "calc" }

process CALC is
let
  Calc : calc,
  E : str,
  V : term
in
  execute(calc, Calc?) .
  ( rec-msg(calc, expr(E?)) .
    snd-eval(Calc, expr(E)) .
    rec-value(Calc, V?) .
    snd-msg(calc, expr(E, V))
  ) * delta
endlet
```

Figure 2.4: The CALC process

2.6 The Connection between TOOLBUS and Tools

In order to connect tools written in an arbitrary programming language, an *adapter* is needed for that language. An adapter is a piece of software that is used to establish connections with the TOOLBUS, and to convert data between the format used internally by the language implementation and the TOOLBUS ATerm format.

Adapters have been developed for a number of languages, including C, Java, Perl, Tcl/Tk, Python, SwiProlog, ASF+SDF, and even COBOL. From the TOOLBUS point of view, there is no semantic difference between tools written in any of these languages, making it the ideal vehicle to construct complex distributed systems using tools written in several languages without suffering from the usual interoperability problems.

Using special purpose adapters often makes it possible to reuse programs off-the-shelf, without even recompiling them. Some examples of the kind of tools we can now use as TOOLBUS tools are the interactive plotting program **gnuplot**, the text editor **emacs**, and the animation tool **samba**.

2.7 Debugging TOOLBUS Applications

An important feature of the TOOLBUS is the built-in *monitor* protocol, which is an extension of the *regular* communication protocol between TOOLBUS and tools. A tool can send two different message to the TOOLBUS: a `snd-value` message as a reply to an earlier `snd-eval` received from the TOOLBUS, and a `snd-event` to signal an event that originated in the tool itself.

A regular tool can receive three different messages from the TOOLBUS, a `rec-eval` message to perform some operation that returns some value back to the TOOLBUS (using `snd-value`), the `rec-do` message when no result is expected back, and the `rec-ack-event` as a indication that a `snd-event` has been processed.

A *monitor tool* introduces three special tool types: the *logger*, *viewer*, and *controller*. A logger tool can be used to log process and tool activity, the viewer tool is a logger tool that can also control the execution speed of TOOLBUS processes, for instance to allow single stepping. The controller tool is a viewer tool that is also capable of changing the internal state of processes.

The current TOOLBUS distribution contains a viewer tool written in the user interface scripting language Tcl/Tk [Ous94]. Using the monitor protocol, the viewer tool is notified of state changes in the TOOLBUS. For instance, whenever a new TOOLBUS process is created, the viewer tool is informed of the name and process-id of the new process.

The standard viewer tool implements a debugging interface consisting of a source window and a TOOLBUS window. Figure 2.5 shows the viewer while stepping through the calculator example discussed in Section 2.5.

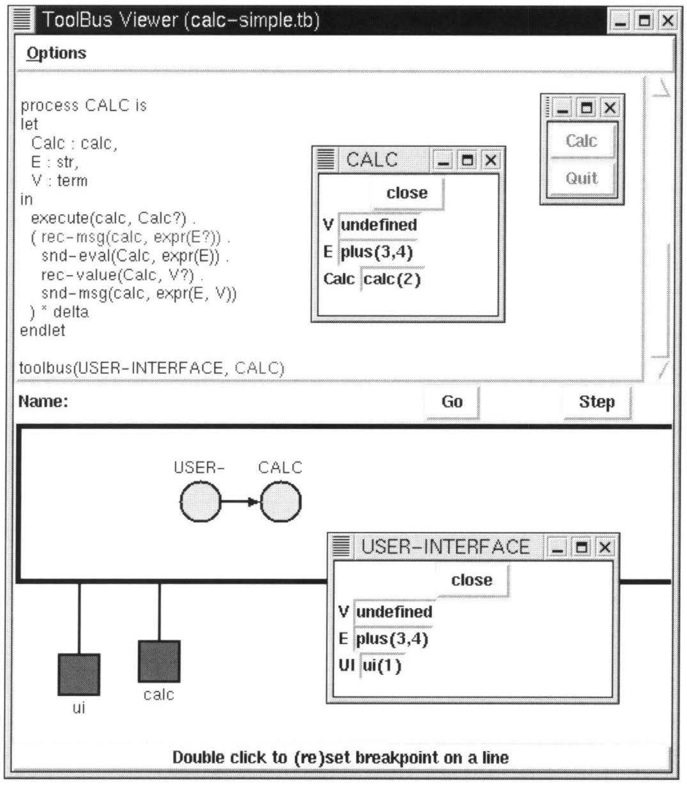
In the source window, the TOOLBUS script being executed is displayed, and actions that are executed are highlighted. The user can run or step through the execution of processes, and can set breakpoints by double clicking on a specific source line. In the TOOLBUS window a picture is drawn of the current processes and tools connected to the TOOLBUS. In this picture, arrows are drawn indicating communication taking place between tools and processes. Double clicking on a process in the TOOLBUS window opens a new window containing the variables of the selected process and their current values.

Although the debugging approach described in this section has proved very useful, it has two major drawbacks:

1. The main debugging component, the viewer, is a monolithic piece of Tcl/Tk code and is therefore hard to extend.

2. Only the external communication behavior of the tools can be observed. The tools connected to the TOOLBUS are treated as *black boxes*. There is no way to inspect the local state of tools.

We will show that using the generic debugging techniques described in this thesis, we can eliminate these problems.


ToolBus Viewer (calc-simple.tb)
Options
Calc
Quit
close
Name:
Go
Step
USER-
CALC
ui
calc
USER-INTERFACE
close
Double click to (re)set breakpoint on a line

```
process CALC is
let
  Calc : calc,
  E : str,
  V : term
in
  execute(calc, Calc?) .
  ( rec-msg(calc, expr(E?)) .
    snd-eval(Calc, expr(E)) .
    rec-value(Calc, V?) .
    snd-msg(calc, expr(E, V))
  ) * delta
endlet

toolbus(USER-INTERFACE, CALC)
```

CALC

```
V    undefined
E    plus(3,4)
Calc calc(2)
```

USER-INTERFACE

```
V  undefined
E  plus(3,4)
UI ui(1)
```

Figure 2.5: Debugging the calculator demo using the viewer

2.8 The TOOLBUS as a component/coordination architecture

In this section, we will compare the TOOLBUS to various *component architectures* and *coordination architectures*. While the term *component architectures* originates from the software industry, the term *coordination architecture* originates from the academic world. Although both types of architectures focus on composition of software components to build distributed systems, their foundations are different. As can be expected, component architectures focus on economic advantages of using components: increased productivity of software engineers and programmers, better project control and easier software reuse. Coordination architectures tend to focus on more 'academic advantages' like theoretical foundations of the architectures, expressiveness of formalisms and the theoretical possibility to prove certain properties like correctness, compositionality and real time characteristics. Because of these different viewpoints, components in component architectures are often quite large, while components in coordination architectures tend to be more fine grained.

We will discuss a number of component architectures (2.8.1 - 2.8.3), and two coordination architectures (2.8.4 - 2.8.5). The TOOLBUS can be seen as a mixture of the two. As a coordination architecture it originated from the academic world and it has a clear theoretical foundation in the form of process algebra. When we look at the TOOLBUS as as component architecture we find that its components are not very fine grained and software engineering benefits like component reuse and programmer productivity are important issues.

2.8.1 CORBA

CORBA has been developed by the Object Management Group (OMG). This consortium set out to solve the problem of interaction between object-oriented systems implemented in different languages and running on different platforms. The solution they came up with was the *Common Object Request Broker Architecture* (CORBA).

The target of CORBA was to make it possible to connect a wide variety of languages, implementations, and platforms. This ambitious goal has one major downside. Individual CORBA implementations cannot talk to each other on an efficient binary level, but must communicate using high-level protocols.

CORBA essentially offers a form of portable remote method calls. As such, it offers a much cleaner model than traditional techniques based on remote procedure calls or even lower level abstractions like direct socket communication. A number of object service specifications (CORBAservices) try to lift this level of abstraction even higher, by standardizing support for high-level services that are important for enterprise level applications. Included are services that provide a security mechanism, object persistence, a transaction mechanism, support for change management, concurrency, and an event notification service. Meta-information can be specified using the CORBA Interface Description Language (IDL).

Note that CORBA does not offer any sophisticated memory management support. This means that without any standardized way to solve these problems, ad hoc solutions will be used in practice to avoid memory leaks.

2.8.2 DCOM

DCOM is the distributed version of the Microsoft's Common Object Model (COM), which has evolved from the OLE (Object Linking and Embedding) framework on Microsoft Windows platforms.

COM is first and foremost an efficient, low-level binary standard. A component in COM consists of a number of *interfaces*, each containing a table with function pointers (a *vtable*). Data is transfered between components on different platforms using a common representation (Network Data Representation, NDR). COM offers a security mechanism, and a complicated persistence mechanism. Meta-information is specified using the COM Interface Description

Language (IDL)[1], and then compiled into *type libraries*.

If we try to look at DCOM as a component architecture instead of a wiring standard, we find that much of its features are needlessly complex, partly because of the emphasis on efficiency, and partly because the DCOM standard has evolved from a complex platform specific framework. Typical examples of this complexity are the error prone way in which distributed memory management is organized using reference counting, and the use of 'dispatch' and 'dual' interfaces to make it possible to call component methods in a generalized matter.

2.8.3 Java based component architectures

The Java success story is also extending towards component software. Any component architecture that is based on 'pure' Java can automatically use standard Java features like introspection to retrieve meta information and serialization to store persistent objects.

JavaBeans The JavaBeans standard introduces a very lightweight approach to component software. The standard is aimed at creating small to medium-sized controls and is therefore not very suitable as a 'general purpose' component architecture. However, the JavaBeans architecture is general enough to be used as a base layer for the InfoBus component architecture discussed below.

RMI The Java Remote Method Invocation standard (RMI) can be used to call methods across Java virtual machines and across networks. It can be seen as a component architecture, as it offers features like object copying across network connections, a naming service to find remote objects, and most important, fully distributed garbage collection.

InfoBus The InfoBus architecture is based on the concept of the *information bus* metaphore, not unlike the TOOLBUS. Objects can plug into this information bus, and listen to events they are interested in. The InfoBus is built on top of the JavaBeans standard, and adds standardization of data transfer between JavaBeans.

JavaSpaces The JavaSpaces architecture offers an interesting approach to component software. Instead of being based on a *message passing* or *remote procedure call* paradigm, JavaSpaces is based on a *shared memory* paradigm. It is also interesting because it is an example of academic cross-fertilization. Many of the ideas behind JavaSpaces are directly derived from the Linda coordination architecture [Gel85].

In the JavaSpaces architecture, components communicate using a number of *object spaces*. These object spaces represent an abstract form of shared memory. Each component can put objects into such an object space, inspect them, and remove objects from an object space.

The JavaSpaces implementation relies heavily on RMI for communication between components and object spaces.

[1]COM's IDL is completely unrelated to CORBA's IDL.

2.8.4 Linda

Most coordination and component architectures are based on message passing. In contrast, Linda [Gel85] is a coordination architecture that is based on the shared memory paradigm. All components communicate using a central 'tuple space'. Linda offers a small number of primitives to place tuples in this tuple space, inspect tuples, and to take tuples out of the tuple space again. Reading tuples is done using matching.

2.8.5 Manifold

Most coordination languages aim at making a clear separation between communication and computation. Manifold [Arb96] does this by separating components (processes) into two categories: *worker* processes and *manager* processes. Worker processes perform computations on data transferred through channels connected to the input and output ports of the processes. Manager processes control the creation and destruction of processes and channels.

2.8.6 TOOLBUS strengths

One common characteristic among the previously discussed component architectures is that the medium used by components to communicate is fixed. CORBA, COM, and Java's RMI all use remote method invocation. The only thing configurable in such architectures is which components are actually connected. The InfoBus offers a couple of different data exchange methods like a subscription and a broadcast mechanism. JavaSpaces offer the object spaces to communicate.

The TOOLBUS on the other hand can be *programmed* using process descriptions in the form of `T-scripts`. The possibility to express the coordination logic of a distributed application in a special purpose coordination layer offers a number of advantages.

Because the component interaction is described at a high level of abstraction, language constructs can be used that are well suited for this task. A typical example of this kind of construct is the non-deterministic choice operator (+). Not many languages have a use for this kind of operator, but it is very useful when describing the behaviour of (communicating) processes.

The TOOLBUS enforces that the coordination logic is "pulled out" of the components, having the effect of making them more general. This in turn promotes software reuse at the component level. As one author puts this *"The* TOOLBUS *promotes the use of generic solutions to specific problems"* [dJ99]. It is not uncommon that a set of TOOLBUS tools is so generic that a number of different applications can be constructed by only changing the TOOLBUS script that connects them.

The use of a specialized language to describe the possible interactions between tools has another major benefit. Much of the "meta-information" that needs to be specified manually with the other architectures, for instance using some IDL, can be derived automatically in the TOOLBUS case. Automatic derivation of component interface descriptions is a major strength of the TOOLBUS, and actually changed the way we design and implement software. One of the most important deliverables of the design process is now a fully functional `T-script` that acts as an "executable design specification". Based on this script, a set of tool interfaces is generated that provide an excellent starting point for the implementation of the individual tools. When the initial design changes, the `T-script` needs to be changed as well. In this case the tool interfaces are regenerated, and the tools can be adapted to fit their changed interfaces.

When compared to the TOOLBUS the other component architectures operate at a lower level of abstraction. Except for the JavaSpaces architecture, the other approaches can be characterized as "wiring standards" [Szy97]. If a developer wants to isolate the communication behavior of an application on a higher level of abstraction he is forced to introduce one or more controlling components that regulate this communication. But because there is no standard approach for this, each developer has to reinvent the wheel in this area and different approaches are incompatible. The TOOLBUS architecture offers exactly this: a central component that controls the communication between other components in a system. This in combination with a specialized language to describe this communication offers a powerful approach to tackle the intricate problem of developing heterogeneous distributed applications.

2.8.7 TOOLBUS weaknesses

When we compare the TOOLBUS to the previously discussed component architectures, we find weaknesses in two categories: performance and functionality.

Performance problems The TOOLBUS performance is lacking because of two major design decisions. The first decision is the choice for a centralized approach. Conceptually, all communication between components and therefore all data transfers go through the TOOLBUS. This means that in many distributed applications the TOOLBUS will be the bottleneck. In order to lift this bottleneck while keeping the conceptually simple view of a centralized mediating bus intact, it will be necessary to transparently route the actual data communication directly between tools. Future research is needed to find out if it is possible to use static or dynamic `T-script` analysis for this purpose.

The second design decision that influences the performance is the use of a common data exchange format in the form of ATerms. Most other component architectures use a similar approach to exchange data between components located on different hosts or in different processes on the same host. But in the TOOLBUS case, ATerms are used to exchanged data even in the case where components are located in the same operating system level process.

Lacking functionality If we make a pass over the features offered by the other component architectures, we can quickly identify some important areas in which the TOOLBUS is lacking support:

- Distributed garbage collection

- Transactions
- Security
- Persistency

Most of these shortcomings can be overcome by programming the support using the `T-script`, in combination with some specialized tools. This has two drawbacks however. First, a `T-script` should be a relatively small and clear description of the communication allowed between components. Adding support for some of the missing TOOLBUS features might reduce the amount of clutter in `T-scripts` significantly.

Secondly, because this kind of support is not standardized, each TOOLBUS programmer has to "reinvent the wheel". In cases where support from the components is required, different solutions might not be compatible with each other. This in turn would make component reuse more difficult. This suggests that it would be beneficial to develop standard solutions for these problems and distribute these standard solutions with the TOOLBUS.

2.8.8 Enlarging the TOOLBUS application domain

To widen the application domain of the TOOLBUS architecture, the efficiency has to be improved drastically. We believe that static and dynamic analysis of `T-scripts` can be used to transparently break the centralized nature of the TOOLBUS. This could lead to an architecture whose efficiency is comparable to that of CORBA. The DCOM approach of using direct method calls when working with in-process components will be hard to match, but comparable performance can be reached in the case where both approaches need to use interprocess communication.

Standardization of features like distributed garbage collection, transactions, a security mechanism, and persistence is important to increase the reusability of TOOLBUS tools. Another important area in which this kind of standardization can have a major impact is the area of *compound documents*. This includes support for things like *drag and drop*, some *clipboard mechanism*, and a standardized way to mix document content from different components.

3
Case study: A Simulator Framework for Embedded Systems

For this case study we have investigated the use of the TOOLBUS coordination architecture for the development of an embedded system simulator. The simulator developed in this case study is a realistic TOOLBUS application of moderate size (over 25,000 lines of code) that has been used in practice to debug applications for a commercially available embedded system. This case study shows that the TOOLBUS provides an excellent architecture for this kind of application. This simulator is also the first TOOLBUS application in the domain of interactive debuggers, providing us with valuable experience for the work described in the next chapter.

3.1 Introduction

In this chapter, we use the term "embedded system" to identify any computer system for which the primary development tools do not run on the system itself. Typical examples are the computer systems that are built into domestic appliances, cars, and airplanes.

Embedded systems that are sold in large quantities often have excellent development support. State of the art simulators, in-circuit emulators (ICE), etc. are available to aid developers.

But many embedded systems are created for a low volume market, and therefore only a few developers are responsible for the software development of these systems. The overhead of developing tools for these systems using 'conventional' methods is just too high. Consequently, these developers still need to rely on print statements and sometimes even memory dumps to debug their software.

It is clear that these developers would benefit enormously from specialized simulators that would boost their productivity and increase the software complexity they can cope with. Programming an embedded system is often a time and money consuming process, and developing and debugging embedded applications can be done much more cost effectively using a simulator instead of using the real hardware all the time.

Dimensions	- $130 \times 100 \times 30mm$.
Environmental	- -20 degree Celsius to +70 degree Celsius.
	- Humidity 40% to 90%.
	Unit is dust and water resistant.
Power supply	- 11 Vdc to 29 Vdc.
Microcontroller	- Hitachi H8/532 16 bit, running at 10 Mhz.
Boot ROM	- 32 Kb.
Flash EPROM	- standard 128 Kb, max. 475 Kb.
RAM	- standard 32 Kb, max. 512 Kb (Field upgradable).
Retention	- 10 year backup with maximum memory.
Communication Ports	- 2 RS 232C DTE serial ports.
	1 RS 232C DTE/DCE programmable port.
Connector	- 9 Pins sub-D.
Baudrates	- 19200, 9600, 4800, 2400, 1200 and many in between.
Format	- 1 start bit, 5-8 data bits, even or odd parity bits,
	- 1 or 2 stop bits.

Table 3.1: UPI-10 hardware specifications

This implies that the effort needed to create a sophisticated simulator should be sufficiently small to warrant its creation. This is only possible if we can achieve a significant amount of reuse, both in design and coding efforts.

This is why we deployed the TOOLBUS. This software coordination architecture was designed to control a number of heterogeneous components in a distributed environment. The TOOLBUS enforces formalization of the communication behavior between the components of the system, making the interaction between them very explicit. When exploited wisely, this can lead to a set of loosely coupled tools with a well defined input/output behavior, greatly improving reusability. The TOOLBUS also provides us with the opportunity to implement each component in the most appropriate language.

The ideal simulator provides all the information about the state of the running program the user wants, and no more. To provide this information in a clear and concise way, a good and extensive user interface is a must. The considerable effort that must be invested into the creation of appealing and easy to use user interface components, logically focussed our attention on making these components reusable.

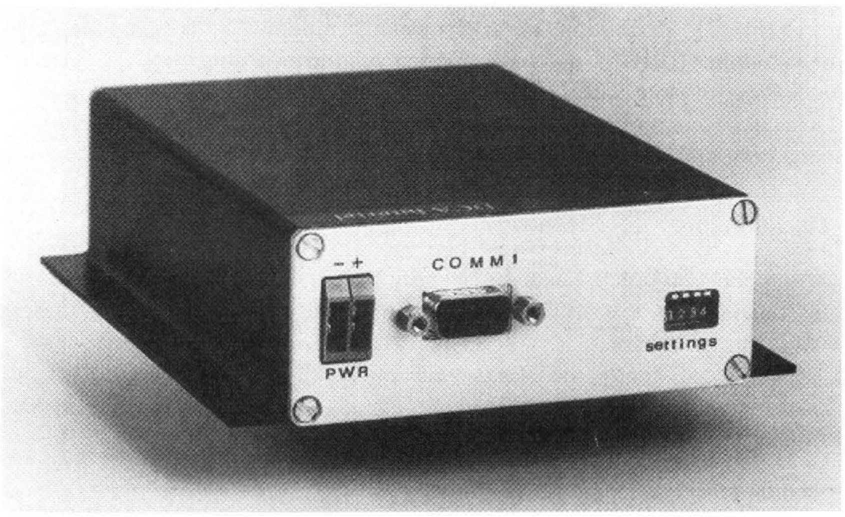


Figure 3.1: The UPI-10

3.2 The UPI-10 Project

There are a lot of devices on the market that are equipped with a serial port for external communication. Examples include terminals, (radio-)modems, mobile printers, lcd displays, mobile phones, etc. Although the majority of these devices use the RS232 protocol at the lowest level, no consensus exists about which higher level protocol to use. Consequently, most manufacturers chooses a protocol best suited for the applications they are interested in. Examples include the vt100 and ANSI escape codes for terminals, and a whole range of protocols for mobile communication, like MOBITEX, MAP 27, and RD LAP Ardis. The UPI-10 (Universal Protocol Interface) was designed as a universal way to interconnect these devices. Figure 3.1 shows a picture of the UPI-10.

The UPI-10 offers three serial ports, a powerful microcontroller, 128Kb Flash ROM and 32Kb RAM. This makes it possible to interconnect three completely different RS232 devices[1] and program the UPI-10 to interface between the different protocols used by these devices.

3.2.1 The Software

A piece of hardware like the UPI-10 is useless without a suitable development environment. Because the UPI-10 was meant to be a low cost product, using third party tools was not an option. We developed an assembler, a linker and a back end for *lcc*, a retargetable C compiler

[1]Even more when a serial multiplexer is used.

described in [FH95]. In addition, we implemented a runtime library containing a subset of the ANSI-C routines ([KR88]) and some communication protocols.

The only thing missing was a powerful simulator. The project described in this chapter resulted in such a simulator that is used to debug and test both the development tools as well as the actual programs developed for the UPI-10.

3.2.2 The Simulator

Before turning our attention to the architecture of the simulator, we will first give an impression of the functionality it has to offer. Figure 3.2 gives an overview of the user interface components of the simulator.

After starting the simulator, the user is confronted with the main *control center*, shown in the center of Figure 3.2. The user can load executables and/or symbol tables and start up the other components, which can be divided into three categories:

- Assembler level debugging components.
- Source level debugging components.
- Communication debugging components.

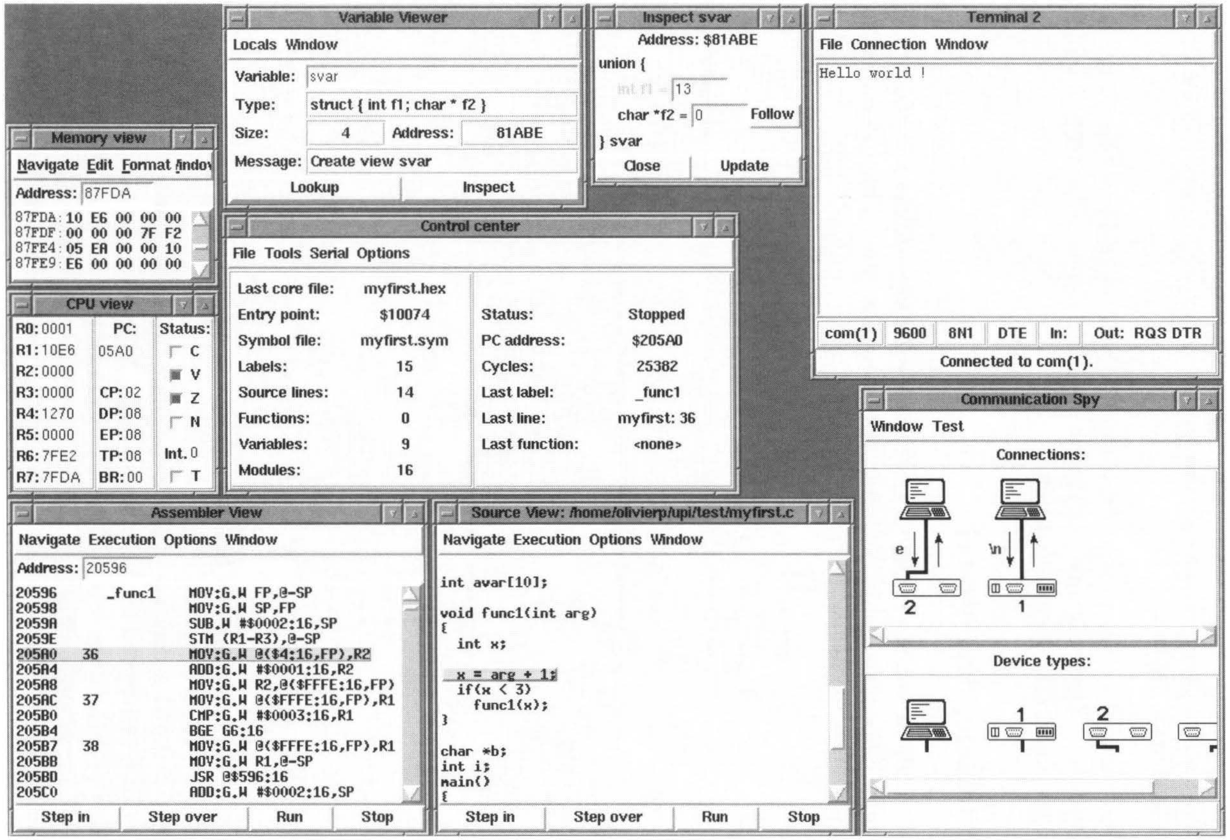
The assembler level debugging components can be used to inspect and interact with the system at the level of the CPU. The components are shown on the left hand side of Figure 3.2:

- *The memory viewer* is used to inspect and change the contents of the memory of the simulated UPI-10.
- *The CPU viewer* enables the user to inspect and change the contents of the CPU registers.
- *The assembler viewer* displays the disassembled memory contents. It allows the user to step through the assembler code and toggle breakpoints.

The source level debugging components handle debugging at the source code level. These components are shown in the middle of Figure 3.2:

- *The source viewer* tracks the current source file and the current point of execution within this file. Breakpoints are highlighted and can be changed.
- The *variable viewer* tracks the contents of variables and lets the user change them.

The communication components simulate the communication between the UPI-10 and any connected RS232 devices. So far, we have implemented two of these components, both of which are shown on the right hand side of Figure 3.2:



Variable: svar
Type: struct { int f1; char * f2 }
Size: 4 Address: 81ABE
Message: Create view svar

```
Address: $81ABE
union {
  int f1 = 13
  char *f2 = 0
} svar
```

```
Hello world !
```

com(1) | 9600 | 8N1 | DTE | In: | Out: RQS DTR
Connected to com(1).

```
Address: 87FDA
87FDA: 10 E6 00 00 00
87FDF: 00 00 00 7F F2
87FE4: 05 EA 00 00 10
87FE9: E6 00 00 00 00
```

```
R0: 0001    PC:      Status:
R1: 10E6    05A0     C
R2: 0000             V
R3: 0000    CP: 02   Z
R4: 1270    DP: 08   N
R5: 0000    EP: 08
R6: 7FE2    TP: 08   Int. 0
R7: 7FDA    BR: 00   T
```

```
Last core file:  myfirst.hex
Entry point:     $10074      Status:         Stopped
Symbol file:     myfirst.sym PC address:     $205A0
Labels:          15          Cycles:         25382
Source lines:    14          Last label:     _func1
Functions:       0           Last line:      myfirst: 36
Variables:       9           Last function:  <none>
Modules:         16
```

```
Address: 20596
20596       _func1     MOV:G.W FP,@-SP
20598                  MOV:G.W SP,FP
2059A                  SUB.W #$0002:16,SP
2059E                  STM (R1-R3),@-SP
205A0   36             MOV:G.W @($4:16,FP),R2
205A4                  ADD:G.W #$0001:16,R2
205A8                  MOV:G.W R2,@($FFFE:16,FP)
205AC   37             MOV:G.W @($FFFE:16,FP),R1
205B0                  CMP:G.W #$0003:16,R1
205B4                  BGE G6:16
205B7   38             MOV:G.W @($FFFE:16,FP),R1
205BB                  MOV:G.W R1,@-SP
205BD                  JSR @$596:16
205C0                  ADD:G.W #$0002:16,SP
```

```
int avar[10];

void func1(int arg)
{
  int x;

  x = arg + 1;
  if(x < 3)
    func1(x);
}

char *b;
int i;
main()
{
```

Figure 3.2: An overview of the simulator

- The *communication spy* keeps track of all the connections established in the system and shows a nice picture of these connections, together with a list of all the device types present in the system.
- The *terminal emulator* is the only RS232 device simulator we developed so far. It emulates a simple terminal that can be connected to one of the UPI-10 serial ports or to another RS232 device simulator.

3.3 Simulator Architecture

Because we aim to create a simulator system that is highly modular and contains a number of reusable components, the basic architecture should be general enough to handle a wide variety of computer systems.

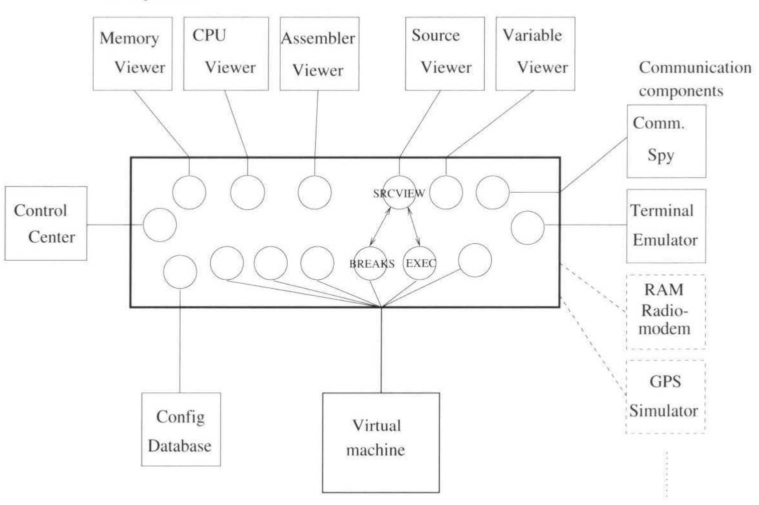


Figure 3.3: Simulator design: component overview

Figure 3.3 shows a diagram of the architecture of our simulator framework. It shows the TOOLBUS surrounded by the different tools. Within the TOOLBUS, the processes are depicted. It is beyond the scope of this thesis to discuss the whole system in detail, so we will outline the general architecture and only present the processes `SRCVIEW`, `BREAKS`, and `EXEC` (see Figure 3.3) in more detail. These processes form a small but coherent subsystem and give a good impression of the kind of communication patterns the system is based on. A more extensive presentation of the internals of the system can be found in [Oli96a].

3.3.1 Virtual Machine

Central to the architecture is the notion of a *virtual machine*. This component has no user interface, but takes care of the actual execution of simulated programs. The virtual machine must simulate all parts of the system that have to run at full speed, so no TOOLBUS communication is needed during the high-speed execution of a program section.[2] To accomplish this, the virtual machine has to keep track of the following information:

- CPU register contents.
- Memory contents.
- The status of any simulated special hardware needed during execution, like switches, LED's, communication hardware etc.
- All breakpoint activity.
- Debug symbols (to map addresses to line numbers etc.).

In addition to the 'bookkeeping' tasks needed to maintain this information, the virtual machine performs the following operations:

- Simulate actual execution of machine language instructions.
- Disassemble the contents of memory on request.

All the other tools, except for the *configuration database*, provide views on the different types of information maintained by the virtual machine, and enable the user to manipulate this information.

The configuration database is used to maintain sets of options, so every user can configure the system to suit his/her preferences.

3.3.2 The Source Viewer Subsystem

We will now present a coherent part of the simulator's TOOLBUS script that clearly demonstrates the system's organization. This subsystem is formed by the three processes named explicitly in Figure 3.3, and describe the communication between the virtual machine and the source viewer.

The `EXEC` process handles all communication directly related to the execution of simulated programs. Its tasks are:

- Starting and stopping the execution of the virtual machine.

[2]We use the term 'program section' in this context to indicate a portion of the program to be executed without a need for visual feedback to the user. For instance, execution of the code generated by one line of source code or by an entire routine depending on the command issued by the user.

- Retrieving the current execution status of the virtual machine (either `running` or `stopped`).
- Catching line-number events generated by the virtual machine, and broadcasting them as notes to the rest of the system.

This is expressed in the following TOOLBUS script:

```
process EXEC(Vm : vmtool) is
let
  Level  : term,
  Action : term,
  Status : term,
  Module : str,
  Line   : int
in
  ( %% Handle requests to start execution.
    rec-msg(exec, Level?, Action?) .
    snd-do(Vm, exec(Level, Action))
    +
    %% Handle requests to stop execution.
    rec-msg(exec, user-break) .
    snd-do(Vm, user-break)
    +
    %% Handle status information requests.
    rec-msg(exec, get-exec-status) .
    snd-eval(Vm, get-exec-status) .
    rec-value(Vm, exec-status(Status?)) .
    snd-msg(exec, exec-status(Status))
    +
    %% The virtual machine can inform us that the current
    %% line number has changed.
    rec-event(Vm, line-number(Module?, Line?)) .
    snd-note(line-number(Module, Line)) .
    snd-ack-event(Vm, line-number(Module, Line))
  ) * delta
endlet
```

The process `BREAKS` handles all communications related to the status of breakpoints. Its tasks are:

- Setting, clearing, and toggling of breakpoints.
- Retrieving the list of all breakpoints maintained by the virtual machine.

- Informing everyone of changes in the list of breakpoints maintained by the virtual machine.

This is expressed in the following TOOLBUS script:

```
process BREAKS(Vm : vmtool) is
let
  Adr            : int,
  Breaks         : list,
  Module         : str,
  Line           : int
in
 ( %% Individual breakpoint toggling.
   rec-msg(breaks, toggle(Adr?)) .
   snd-do(Vm, toggle-break(Adr))
   +
   %% Turn of all breakpoints related to one source line.
   rec-msg(breaks, line-off(Module?, Line?)) .
   snd-do(Vm, line-break-off(Module, Line))
   +
   %% Turn on all breakpoints related to one source line.
   rec-msg(breaks, line-on(Module?, Line?)) .
   snd-do(Vm, line-break-on(Module, Line))
   +
   %% Handle requests for the list of current breakpoints.
   rec-msg(breaks, get-breaks) .
   snd-eval(Vm, get-breaks) .
   rec-value(Vm, breakpoints(Breaks?)) .
   snd-msg(breakpoints, breakpoints(Breaks))
   +
   %% When a change has taken place in the list of
   %% breakpoints, the complete list is broadcasted.
   rec-event(Vm, breaks(Breaks?)) .
   snd-note(breaks-changed(Breaks)) .
   snd-ack-event(Vm, breaks(Breaks))
  ) * delta
endlet
```

On the source viewer side, the `SRCVIEW` process handles all communication, its tasks are:

- The creation of new source-viewer windows.
- Informing the source-viewer of changes in the execution status of the virtual machine.

- Informing the source-viewer of changes in the list of breakpoints maintained by the virtual machine.
- Receiving requests to turn on/off breakpoints and sending them to the BREAKS process.
- Receiving requests to start or stop execution, and sending them to the EXEC process.

This is expressed in the following TOOLBUS script:

```
process SRCVIEW is
let
  Src    : srcview,
  Level  : term,
  Action : term,
  Module : str,
  Line   : int,
  Breaks : list
in
  %% Start the source viewer tool.
  execute(srcview, Src?) .
  %% Subscribe to some interesting note types.
  subscribe(line-number(<str>, <int>)) .
  subscribe(breaks-changed(<list>)) .
  subscribe(config-changed) .
  ( %% Handle source-viewer window creation requests
    rec-msg(srcview, create-view) .
    snd-do(Src, create-view)
    +
    %% The program counter has reached a new source line.
    rec-note(line-number(Module?, Line?)) .
    snd-do(Src, line-number(Module, Line))
    +
    %% The breakpoint information has changed.
    rec-note(breaks-changed(Breaks?)) .
    snd-do(Src, breaks-changed(Breaks))
    +
    %% The user wants to set a breakpoint. Propagate the
    %% request to the BREAKS process.
    rec-event(Src, break-line-on(Module?, Line?)) .
    snd-msg(breaks, line-on(Module, Line)) .
    snd-ack-event(Src, break-line-on(Module, Line))
    +
    %% Idem for turning off a breakpoint.
    rec-event(Src, break-line-off(Module?, Line?)) .
    snd-msg(breaks, line-off(Module, Line)) .
    snd-ack-event(Src, break-line-off(Module, Line))
    +
    %% Handle a request to continue the programs execution.
    rec-event(Src, exec(Level?, Action?)) .
    snd-msg(exec, Level, Action) .
    snd-ack-event(Src, exec(Level, Action))
    +
    %% Stop the program execution.
    rec-event(Src, user-break) .
    snd-msg(exec, user-break) .
    snd-ack-event(Src, user-break)
  ) * delta
endlet
```

The whole interconnection layer of the system consists of these kinds of simple TOOLBUS scripts that describe the communication patterns between components. Studying these well organized TOOLBUS scripts makes it possible to gain a high level understanding of a system without needing to know all implementation details. In cases where more detail is needed, the source code of the different tools can be consulted.

3.4 Conclusions

In this section, we will present some of the conclusions that can be drawn from this case study.

3.4.1 Constraints

A system has to satisfy several constraints in order to fit in this simulator framework. When faced with a project that might benefit from this work, these constraints might be a good place to start your evaluation.

Hardware architecture The framework is biased towards what one might call a *classic* architecture. This means a system containing a single processor[3] that reads its instruction and data streams from a central store. The simulated processor must at least implement a program counter so the other tools can track the execution and a frame pointer if there are any stack based (local) variables.

[3]We do not exclude a multi-processor environment, but this would require some work.

	Basic tool				
VM tool provides:	Control Center	CPU Viewer	Memory Viewer	Source Viewer	Variable Viewer
static features					
address labels	†	†	†		
line number info	†	†	†	‡	
function info	†				
variable info					‡
scope info					‡
dynamic features					
program counter	†	‡		‡	†
cycle counter	†				
frame pointer					†
cur. scope list					†
breakpoints		†		†	

Table 3.2: Simulator dependencies

Software support To be able to make sense of the contents of the simulated memory and register contents, the simulator needs static symbol information. In most cases this information is generated by the linker. Table 3.2 shows the information needed and the tools that depend on it. A dagger (†) in a certain column means that that particular tool *uses* the feature mentioned in the first column, but still works without it (albeit with some reduced functionality). A ‡ means that a tool *depends* on a feature and is pretty useless without it.

3.4.2 Evaluation

How well did we achieve the reusability and performance targets mentioned in the introduction of this chapter?

Reusability Targets The reusability of the system is best illustrated by some statistics about the size of the software (Table 3.3). We focus on reusability of components when using different CPU's. Typically, compiler backend, assembler, linker, and virtual machine will then have to be replaced. The architecture of the simulator makes it easy to add new components when other elements change besides the CPU.

Of course, Table 3.3 only provides a very rough estimate, mainly because the implementation languages are so diverse.

An obvious conclusion is that the simulator size is almost as big as the other development tools together, warranting a major investment to reuse parts of it.

UPI-10 development tools				
	Component	Predominant language	Lines of code (reusable)	Lines of code (not reusable)
development tools	lcc front end (pre-existing)	C	13741	
	lcc back end	C		2546
	assembler	C++		5477
	linker	C++		2502
	library	Assembler		5632
	Total dev. tools		13741	16157
simulator	virtual machine	C++		17420
	UI components	Tcl/Tk	8252	
	communication control	TOOLBUS script	1657	
	Total simulator		9909	17420
	Percentage reused		36%	64%

Table 3.3: Some code statistics

In a new simulator, the UI components and the TOOLBUS script are directly reusable. This means an instant reuse of 36 percent of high-level simulator code.

Although the virtual machine is marked as *non reusable*, this is not strictly the case. This project was centered around the reuse of complete tools. Because the virtual machine is nothing but a very specialized program, traditional techniques aimed at reusability can be applied very successfully.

Performance Targets Because the actual program execution takes place within in the virtual machine, the simulator performance in this area is not influenced by the general simulator architecture. When running the virtual machine on a 100 Mhz Pentium PC, it outperforms the actual UPI-10 hardware so delays need to be introduced to make the behavior more realistic.

Although the user interface components are all separate tools, requiring inter-process communication on almost every user interaction, the interactive response times are quite good, essentially justifying the choice for the TOOLBUS concept.

3.4.3 TOOLBUS Experience

One of the goals of this project was to gain experience with TOOLBUS programming. Now that we have implemented one of the first medium sized TOOLBUS applications, some preliminary conclusions can be drawn.

- The TOOLBUS is very good in separating components, especially if the traffic between them can be kept light.

- An interesting separation technique is the complete separation between the user interface and computational components. Because the TOOLBUS communication is very fast compared to a user interacting with a user interface, only a small performance penalty is paid by separating the components.
- The one-to-one mapping between tools and programs is very unsatisfactory. The need for interprocess communication for every TOOLBUS/tool interaction seriously affects performance.

The only reason why we did not split up the system even further, for instance by separating the symbol table manager and the actual virtual machine, was a lack of performance. This suggests a general strategy when designing a system around the TOOLBUS.

- First start with an (existing?) design where all components are located in one tool.
- Move all user interface components into separate tools.
- Try to isolate other components that can be turned into tools without seriously affecting the overall performance.

Several studies on micro-kernel operating systems (for instance the Amoeba operating system, [RvRT89]) have shown that it is possible to get a decent performance while using message passing. The possibility to incorporate some tools in the TOOLBUS instead of connecting them using interprocess communication primitives will make a big difference and opens up a wide range of application areas.

3.4.4 Future Research

We definitely need to develop more simulators, in order to refine and extend this framework.

An interesting experiment would be to replace the virtual machine tool with a symbol table manager and a program debugging interface to programs running on the *host* machine, so the tool can monitor and interact with these programs. This would extend the framework to incorporate debuggers as well as simulators *without changing the user interface components.* Chapter 4 describes a debugging framework that has been inspired by the work described in this Chapter.

On the TOOLBUS side, a number of interesting fields lay open for investigation:

- How much can we boost performance by moving tools into the TOOLBUS thus eliminating the interprocess communication?
- The dynamic loading of new TOOLBUS scripts seems to be a logical next step, complementing the dynamic creation of processes and the dynamic connection/disconnection of tools, both of which are already present.

- The TOOLBUS script for the simulator grew to considerable size (more than 1,600 lines), but was still manageable. As we gain experience with the TOOLBUS, our scripts will grow and we will need some kind of modularization construct.
- The TOOLBUS uses a string representation of terms to communicate with the tools. This makes it very hard to share common subterms. Experimenting with an architecture independent term or graph representation that preserves sharing would be interesting. In Chapter 6 we show how this kind of sharing can be achieved.

4 Debugging Heterogeneous Distributed Applications

Our experience gained from the work described in the previous chapter convinced us that it is possible to design a generic language independent debugging framework based on the TOOLBUS coordination architecture.
In this chapter, we present a debugging framework for debugging heterogeneous distributed systems. In such systems, a variety of languages can be used for the implementation of individual components, so our debugging framework must be able to deal with these languages. Instead of reinventing the wheel in this area, we try to reuse existing debugging support for these languages as much as possible.

The majority of the existing debuggers for these languages work by abstracting the behavior of the program being debugged into *events*, and visualizing these events. We utilize these sequential debuggers to generate language-independent debugging events related to the sequential execution of the components in the distributed system. The underlying *coordination architecture* (in our case the TOOLBUS) is used to generate debugging events dealing with the interaction between components. These sequential and process interaction related debugging events are then processed by a separate distributed system that implements the high-level language-independent debugging functionality.

4.1 Introduction

Debugging is the process of locating and fixing errors (*bugs*) in software systems. A debugger is a software tool that can help understand a system being debugged by visualizing different aspects of its execution.

Although multilingual sequential debuggers have been around for quite some time [Bea83, SP91], many distributed debugging tools are based on support from a single experimental operating system or language environment [For89].

In the area of *heterogenous* distributed systems, things get even worse. Debugging support is often limited to tracing the communication between components, and the debugging of the components themselves is left to traditional tools for debugging sequential programs. Although most modern language implementations have some kind of debugging support built in, they all have their own interface and command set. Having to work with a number of debuggers *at the same time* can make it very difficult, if not impossible, to debug such heterogeneous systems.

This calls for an interface between distributed and sequential debuggers, in order to combine the two fields [CBM90] and reuse existing implementations. We have designed such an interface, and subsequently built a powerful, multilingual debugger for distributed systems.

Most debuggers work by gathering *primitive events*, filtering or clustering them, and presenting the results to the user. Details on how to gather events, which filtering or clustering algorithms to use, and when and how the results are presented differ in each case. Unfortunately, when it comes to event gathering, every system seems to reinvent the wheel. Solutions ranging from hardware assisted compiler instrumented code [AY91], to manually inserted event generation calls [BW83] can be found in the literature. In this chapter we show that by reusing these low level event gathering implementations, we can leverage the existing plethora of debugger implementations into a single framework for debugging heterogeneous distributed applications.

We have combined the power of the TOOLBUS coordination architecture with existing low level debugging interfaces for sequential programming languages. The resulting framework consists of language-dependent debugging interfaces, based on the native debugging support found in most existing language environments, coupled with language-independent debugging components. By adding event reporting for the coordination architecture we use, the resulting set of primitive events is rich enough to build a solid distributed debugging environment.

To show the feasibility of our approach, we have constructed a debugger for distributed applications. The debugger offers a uniform graphical user interface for inspecting the communication behavior of the system being debugged, and for tracing the source level execution of the components of the system. Our debugger is both extensible in the set of languages it can handle, as well as in the debugging functionality it has to offer, because of the debugging framework it is based on. The design and implementation of this debugger is discussed in Section 4.5.

In Section 4.2 we introduce our basic framework. In Section 4.3 we present the notion of event rules that play a central role in our framework, and in Section 4.4 we show how these event rules can be used to implement some of the functionality that is typically found in traditional sequential debuggers.

4.2 A TOOLBUS Framework for Debugging Distributed Applications

The majority of distributed systems consist of a number of sequential components running in parallel. We present a debugging framework for distributed systems which uses the events generated by native debuggers for these sequential components. These native debugger events

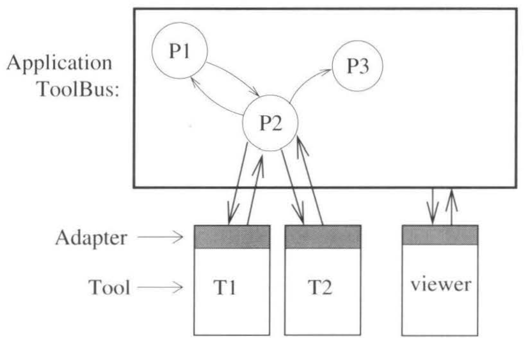


Figure 4.1: TOOLBUS architecture with viewer

are translated into language-independent debugging events. The events generated by the TOOLBUS monitor protocol (see Section 2.7) are used to generate events about the interaction between processes. The combination of language-independent debugging events and process interaction events are used to synthesize high-level debugging facilities not found in the underlying native debuggers. Examples of these facilities are:

- A uniform graphical user interface for debugging all tools, independent of the language they are written in.
- Tracing and animating execution at the source code level, independent of the language the tool is written in.
- Full conditional breakpoint and watchpoint support, based on the expressions discussed in Section 4.3.3, even if the underlying native debugger does not support conditional breakpoints.
- Breakpoints and watchpoints that can be set on any type of event, not just on specific locations in the source code. For example, watch the value of a variable whenever a certain message is sent.

We base our framework on the assumption that the distributed system being debugged is also based on the TOOLBUS coordination architecture. However, our techniques can easily be generalized to other distributed architectures as the ones discussed in Section 2.8.

Let us first recall the basic TOOLBUS architecture from Chapter 2. Figure 4.1 shows how in the standard TOOLBUS architecture a *viewer* tool can be connected to debug TOOLBUS scripts.

The first step towards this new architecture is to introduce a second TOOLBUS based distributed system which actually implements the debugger as shown in Figure 4.2. The TOOLBUS used in the system being debugged will be called the *application bus*, and the one used in the debugging system will be called the *debugging bus*.

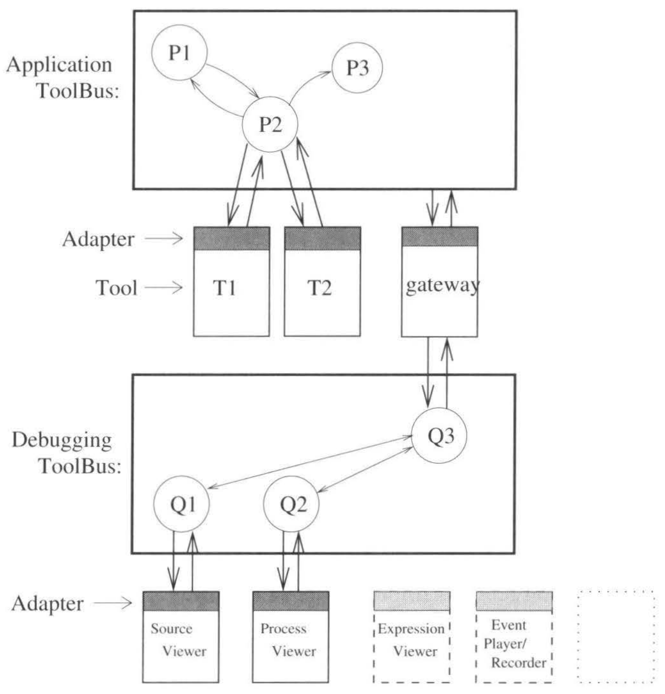


Figure 4.2: Turning the debugger into a distributed system

In Section 2.7 we explained that the viewer tool visualized debugging events to provide the user with visual feedback on the execution of the application bus. In our new architecture, the TOOLBUS viewer tool is replaced by a tool that acts as a *gateway* between the application bus and the debugging bus. This gateway is used to forward the debugging events received from the application bus directly to the debugging bus. The actual debugging functionality is implemented in the debugging bus using a number of cooperating tools, for instance, a source code browser and a process viewer. This architecture makes it possible to break up the complex implementation of the debugger in a number of more manageable components.

But we can take this architecture one step further. To do this, we must realize that most of the languages used to implement the components that are connected to the application bus have some kind of native debugger support. The debugging interfaces of languages like *Java* [GJS96], *Python* [WvRA96], *Tcl/Tk* [Ous94], and *C* (using for instance *gdb*) all offer the possibility to implement a sophisticated debugger on top of a low level debug interface.

Figure 4.3 shows our final architecture, in which the individual components of the application bus are also connected to the debugging bus using a *debug-adapter*. This debug-adapter uses the low level native debug interface to generate debug events related to the actual execution of these components.

It is important to note that components can contain multiple lightweight processes (sometimes called *threads*), for instance in the case of a Java component. As we will show later on, each debug-adapter is responsible for processing the low level debug events of all debuggable processes in the component it is part of.

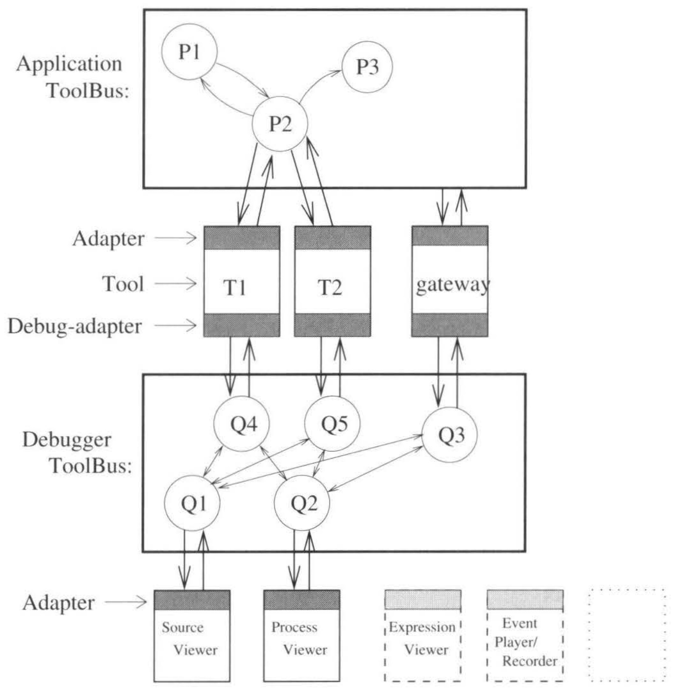


Figure 4.3: Full debug event gathering

4.3 Event Rules

Most debuggers gather primitive events and use these to inform the user of what is going on in a program. Our framework reuses these native debugging events to do the actual event gathering. The primary task of the debug adapters introduced in Section 4.2 is to unify and filter these events in order to make our framework language-independent.

We do this unification and filtering of low level debugging events by using *event rules*. An event rule consists of an *event port*, an *event condition*, and a list of *event actions* (Figure 4.4). The event port indicates at which points during the execution of a program an event rule is *activated*. When an event rule is activated, its event condition is evaluated to see if

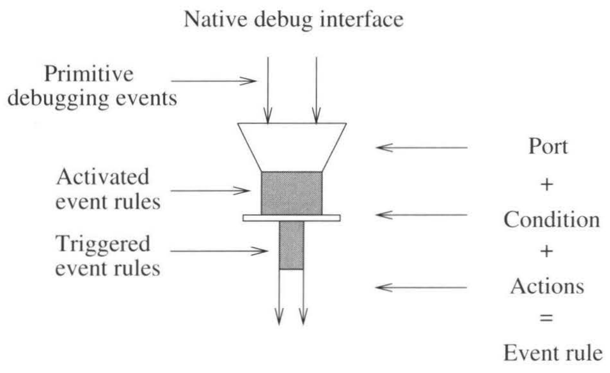


Figure 4.4: Event rules

the event rule should be *triggered*. Triggering the event rule consists of executing its event actions. The event actions can generate unified debugging events handled by the application bus, or they can influence the execution of the component that triggered them, for instance by halting execution (*breakpoints*), or changing the contents of a variable.

Event rules are process specific. It is the task of the debug-adapter to maintain a set of event rules for each process in the component that is being debugged. Note that most components contain only one process. Only multithreaded/multiprocess applications (for instance Java applications or the TOOLBUS) can contain multiple processes.

4.3.1 Required Support

In order to be useful in our framework, the components being debugged must support some basic features. Surprisingly, the set of features required to be able to implement a minimal debug adapter is quite small. Only four features are absolutely necessary:

- The component can consist of multiple processes, but each process must support the notion of a *current point of execution* (*cpe*) in order to be able to debug that process. It must be possible to relate this *cpe* to a location in the source code of the component.
- It must be possible to start and stop the execution of each process in the component.
- It must be possible to single step through the execution of a process. The granularity of such a "single step" must be intuitive at the source code level. Single stepping one assembly instruction is not all that useful when working at the source code level.
- There must be support for breakpoints.

Other features are not absolutely necessary, but are only required to enable certain debugger functionality. For instance, the debugger can only offer the "step over" functionality discussed in the next section when the component being debugged has a stack-based execution model, and the debug-adapter can access the current stack depth. Another example is the possibility to inspect the value of variables. This will only work when the debug-adapter of the component being debugged has some means of retrieving the value of a variable.

Stated differently, our framework offers a form of *graceful degradation*. The fewer the features for the debug-adapter to work with, the fewer debug functionality is offered to the user of the debugger. But the whole mechanism only breaks down when the debug-adapter does not implement the *cpe* or execution control functionality discussed above.

4.3.2 Event Ports

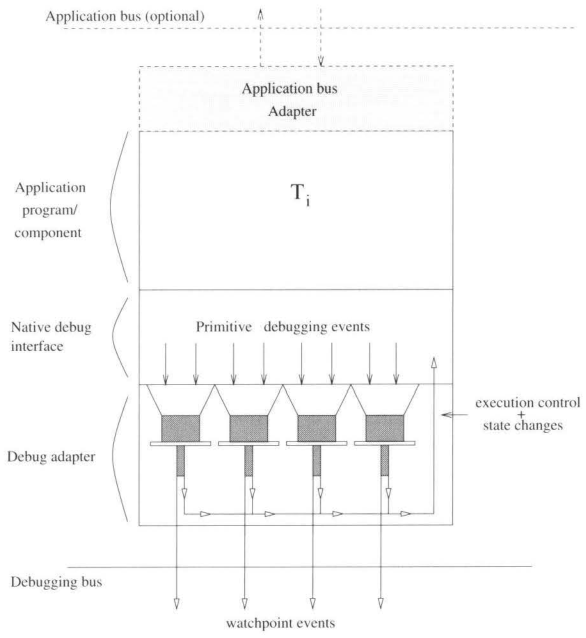

Application bus (optional)
Application bus Adapter
Application program/ component
T_i
Native debug interface
Primitive debugging events
Debug adapter
execution control + state changes
Debugging bus
watchpoint events

Figure 4.5: Event processing inside a debug adapter

The *event port* is used to bind a set of native debugging events to an event rule. Table 4.1 shows a list of event ports we are currently considering. This list can be extended or restricted to suit the needs of a particular implementation[1].

[1]The term *event port* is based on the standard Prolog 4-port tracer [CM87]. The `entry` and `exit` ports are directly based on their Prolog counterparts.

Port	Activated when
`entry`	a function/predicate is called.
`exit`	exit from function/predicate.
`stopped`	execution stops.
`started`	execution continues.
`location(`*loc*`)`	the specific location in the source code indicated by *loc* is reached.
`var-access(`*var*`)`	the contents of a variable is accessed.
`var-change(`*var*`)`	the value of a variable changes.
`exception`	an exception or error occurs.
`step`	a statement is executed.
`send`	a message is sent.
`receive`	a message is received.

Table 4.1: Event ports

4.3.3 Event Conditions

Event conditions are used to filter uninteresting events locally. This prevents the debugging bus from having to handle excessive amounts of events, and protects the user from an excessive amount of information that he or she did not ask for. For instance, by implementing conditional breakpoints even when the underlying native debugger only supports unconditional breakpoints.

The event condition is an expression that is evaluated when the event rule is activated. The event rule is only triggered when this expression evaluates to *true*. The debug adapter decides which terms are considered to be equivalent to *true* and which are not.

In our framework, there are two places where we use expressions:

- In the condition of event rules.
- In the arguments of actions of event rules.

Unfortunately, the syntax and semantics of expressions are different in every programming language. To make things even worse, not every debugger supports the evaluation of these *native* expressions.

Our framework offers the user *mixed expressions* in order to overcome the inadequacy of some debuggers in this area. Our expression language consists of prefix functions that are evaluated whenever their value is needed.

These expressions are called *mixed*, because there is a special function `eval`, which takes a single string argument. This string is passed to the native debugging interface for evaluation, The native debugger determines how this string is interpreted, but most debuggers will allow the user to use the syntax of the language being debugged. Although the result type of most

Function	Description	Debug functionality that depends on this function (see Section 4.5)
`state`	Retrieves the execution state of the current process.	Process status viewing
`cpe`	Retrieve the current point of execution.	Highlighting cpe in the source code
`var`	Retrieves the value of a variable.	Conditional breakpoints and variable viewing
`msg`	Retrieve the last message sent or received.	Communication viewing
`stack-depth`	Retrieve the current depth of the stack.	step-over
`start-depth`	Retrieve the depth of the stack at the time the last `resume` function was executed.	step-over
`true`	Always returns `true`	Basic support
`false`	Always returns `false`	Basic support
`equal(`*`t1`*`,`*`t2`*`)`	Returns `true` when *t1* and *t2* are equal, `false` otherwise.	Basic support
`higher-equal (`*`t1`*`,`*`t2`*`)`	Returns `true` when *t1* is a number that is higher or equal than *t2*.	Basic support

Table 4.2: Functions for use in event conditions and event actions

functions is predefined, the result of `eval` is left to the debug adapter. For instance, if it can determine that the result is of type `integer`, an integer is returned. It can always return a string representing the result when there is no other sensible TOOLBUS term equivalent. Fortunately, most TOOLBUS adapters already provide a generic way to translate internal data objects into TOOLBUS terms.

A number of other functions can be used in expressions, including functions to retrieve the current state of execution of the process being debugged or to retrieve the value of a variable. Mathematical and logical operators are also included to perform simple calculations and comparisons. Tables 4.2 and 4.3 give an overview of the functions that are required to implement all the debug functionality discussed in this chapter. This is not a closed set of functions, but it will expand as more and more functionality is added to an implementation. Not all debug-adapters have to support all of these functions. Some features are just not available when the responsible debug adapter does not support the correct functions.

The event port and event condition can be used by the debug adapter to configure the un-

Function	Description	Debug functionality that depends on this function (see Section 4.5)
`break`	Stop the execution of the current process.	Explicit execution control, breakpoints
`resume`	Resume execution of the current process.	Explicit execution control

Table 4.3: Functions for use only in event actions

derlying native debugger. In this way, events can be filtered in an early stage, using hardware or operating system support when possible. A typical example of this kind of optimization is the use of low level conditional breakpoints to avoid activation of rules when the condition would fail anyway.

4.3.4 Event Actions

When an event rule is triggered, its *event actions* are evaluated. When the evaluation results in a value other than `true`, a generic debugging event is generated that can be processed by the debugging bus.

The evaluation of an event action can also cause a change in the (execution) state of the process that triggered the event rule. It can for instance halt the execution. In this case, the event rule that triggered the action is functionally equivalent to a *breakpoint* in traditional debugging terminology.

All expressions described in Section 4.3.3 can also be used in event actions. In event actions some special actions can also be used that change the (execution) state of a process, as shown in Table 4.3.

4.4 Example Event Rules

In this section, we will show how some well known debugger features can be implemented using event rules. Our debugger implementation (Section 4.5) implements these features using the event rules described in this section.

4.4.1 Single Stepping Execution

One of the most important features of any debugger is the ability to single *step* through the execution of a segment of code.

Ensuring execution stops after executing a single step is done using the event rule:

Port:	`step`
Condition:	`true`
Action:	`break`

After each step, the current location of the program counter is usually visualized by printing or highlighting the appropriate piece of source code. Which event rules are used to perform this highlighting is shown in Section 4.4.3.

4.4.2 Stepping over Function Calls

A variation of the *step* command discussed above, is the *step over* command. When the user issues this command, a single instruction is executed, just as with the *step* command. If this instruction happens to be a function call, the entire call is executed before control is returned to the user.

To implement this, we use the same event rule as with the *step* command, but instead of leaving out a condition, we introduce a condition that uses the depth of the stack to determine whether to halt execution or not:

Port:	`step`
Condition:	`higher-equal(start-depth,stack-depth)`
Action:	`break`

The built-in function `start-depth` returns the depth of the stack when execution was last started, i.e., at the time the last `resume` action was executed. The function `stack-depth` returns the current depth of the stack. Whenever the current depth of the stack equals or exceeds the depth at which execution was started, we know that we are no longer executing a function call and so we can halt execution.

Note that some debug-adapters optimize this event rule by translating it into an explicit 'step over' call in the native debugger. The native debugger in turn kan implement this by placing a breakpoint on the next line in the current function. This way native breakpoint support can be used to execute called functions at full speed instead of activating the event rule after every instruction of that function.

4.4.3 Instruction Highlighting

Most debuggers can visualize the execution of a program by highlighting instructions as they are executed. This instruction highlighting can be found in two flavors:

- Highlight on halt.
- Execution animation.

Both flavors can be implemented using event rules that utilize the function `cpe` that returns the *current point of execution* of a process using coordinates related to the original source code.

To highlight the current instruction whenever the execution is halted, we use the event rule:

Port:	`stopped`
Condition:	`true`
Action:	`cpe`

By changing the port from `stopped` to `step`, the event rule is triggered with the execution of every instruction. This enables the debugger to animate the execution of the process:

Port:	`step`
Condition:	`true`
Action:	`cpe`

4.4.4 Watching Variables

To watch the current value of a set of variables whenever the execution of a process stops, we use the function `var(`*name*`)` that retrieves the current value of a variable. By creating this event rule:

Port:	`stopped`
Condition:	`true`
Action:	`var(`*`variable-name`*`)`

The value of the variable is sent to the debugging bus so it can be displayed whenever execution stops.

We could also change the port from `stopped` to `var-change` to animate the value of the variable during program execution:

Port:	`var-change(`*`variable-name`*`)`
Condition:	`true`
Action:	`var(`*`variable-name`*`)`

4.4.5 Breakpoints

Almost every debugger enables the user to set a breakpoint at a specific location in the source code. When the execution of the program reaches this location, the execution is halted. In our framework, this kind of breakpoint is called 'simple breakpoint'. A simple breakpoint is implemented by creating the event rule:

Port:	`location(loc)`
Condition:	`true`
Action:	`break`

Where *loc* is the desired breakpoint location in the source code given in source coordinates (file name and line number). By simply adding conditions to the event rule, we can implement conditional breakpoints as well.

4.5 Implementation

So far, we introduced a framework for language independent distributed debugging. In this section, we will present an actual debugger we have implemented based on this framework. This implementation shows how the framework can be used as a basis for a generic debugging system that can support a variety of languages and debugging tools. We will give an overview of the features present in this implementation, and give an impression of the user interface of the various debugging tools.

We have named our debugger TIDE: the **T**oolbus **I**ntegrated **D**ebugging **E**nvironment. It offers impressive advantages over 'standard' debugging environments:

- TIDE can debug distributed systems based on the TOOLBUS.
- TIDE operates in any operating environment that is supported by the TOOLBUS. This includes most modern unix operating systems.
- When the distributed system to be debugged is based on the TOOLBUS, TIDE can display a graphical representation of the processes and communication in the system.
- Even in non-TOOLBUS based distributed systems or with stand-alone programs, TIDE can trace sequential components at the source level, as long as the component is implemented using one of the language environments supported by TIDE.
- Support for a new source language only requires a new debug adapter to interface with the TIDE system. Because these adapters are built on top of the native debugging interface of the language implementation in question, it is typically very small (300-1000 lines of source code). We have implemented debugging adapters for the following languages:
  - C
  - Java
  - Tcl/Tk
  - ASF+SDF[2]

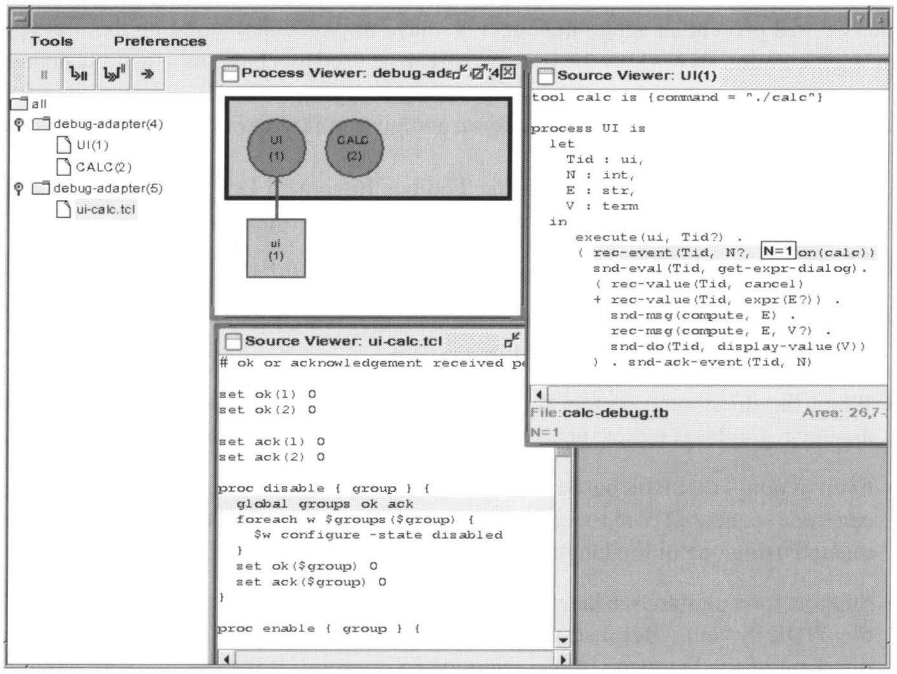
[2]In Chapter 8 we present TIDE debugging support for ASF+SDF.

  - TOOLBUS scripts

- Because of the loose coupling between the debug adapters and the debugging tools, it is also relatively easy to add new debugging tools.

The debugger TOOLBUS is the center of the TIDE system. Connected to it are two kinds of tools: debugging adapters and debugging tools. The adapters generate debugging events related to the execution of the program being debugged. The debugging tools control the configuration of the debugging adapters, and visualize the debugging events.

4.5.1 User interface



```
tool calc is {command = "./calc"}

process UI is
  let
    Tid : ui,
    N : int,
    E : str,
    V : term
  in
     execute(ui, Tid?) .
     ( rec-event(Tid, N?, N=1 on(calc))
       snd-eval(Tid, get-expr-dialog) .
       ( rec-value(Tid, cancel)
       + rec-value(Tid, expr(E?)) .
         snd-msg(compute, E) .
         rec-msg(compute, E, V?) .
         snd-do(Tid, display-value(V))
       ) . snd-ack-event(Tid, N)
```

```
# ok or acknowledgement received pe

set ok(1) 0
set ok(2) 0

set ack(1) 0
set ack(2) 0

proc disable { group } {
  global groups ok ack
  foreach w $groups($group) {
    $w configure -state disabled
  }
  set ok($group) 0
  set ack($group) 0
}

proc enable { group } {
```

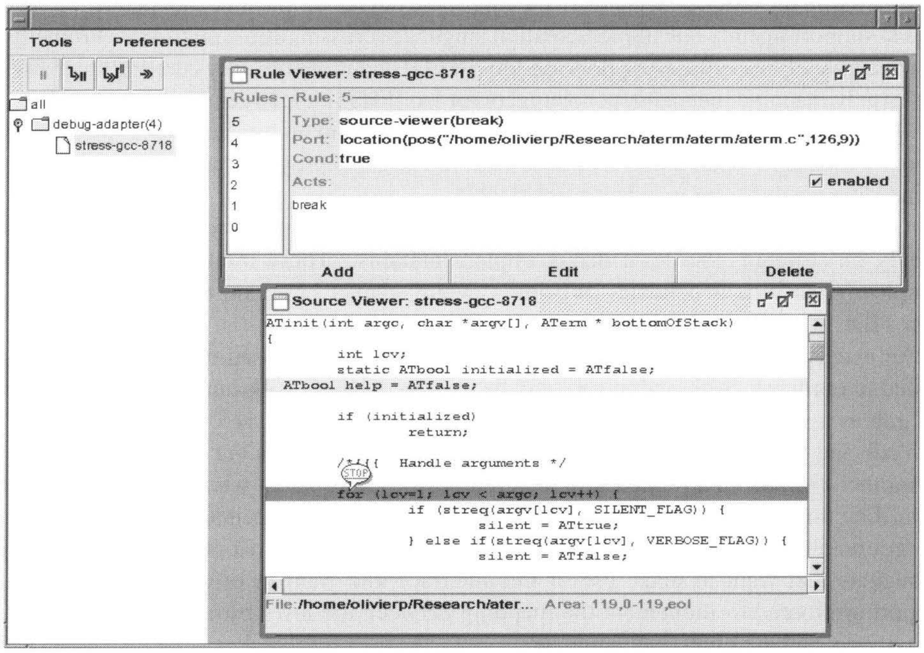
Figure 4.6: Debugging a distributed system using TIDE

Figure 4.6 shows a screenshot of a debugging session. The system being debugged is the *calculator example* presented in Section 2.5. On the left side a list is shown of all debug adapters that are connected to the TIDE system. For each debug adapter, a tree of processes managed by the adapter is shown. Most debug adapters manage only a single process. Exceptions are debug adapters for multithreaded or parallel languages like Java and the TOOLBUS itself. In these cases, each thread or process is handled separately.

Next to this list of debug adapters, the *process viewer* is visible. This debug tool shows an overview of the TOOLBUS system being debugged, complete with processes, tools, and communication between them using arrows.

Below the process viewer, an instance of the *source viewer* shows the source code of one of the tools being debugged, in this case a tool written in Tcl/Tk. The current point of execution in this tool is highlighted.

To the right of the process viewer, another instance of the source viewer is used to display the source code of one of the TOOLBUS processes. Again the current point of execution is highlighted. In this window, the value of the variable N is displayed by right-clicking on it. When right-clicking on a variable, the source viewer retrieves the current value of that variable from the debug adapter, and displays it using a small popup window. This approach has the advantage that the user of the debugger does not have to make a mental 'context switch' to another window to view the value of a variable: the information is readily available at the location where it is needed.



Rule: 5

Rules	
5	Type: source-viewer(break)
4	Port: location(pos("/home/olivierp/Research/aterm/aterm/aterm.c",126,9))
3	Cond: true
2	Acts: [✓ enabled]
1	break
0	

```
ATinit(int argc, char *argv[], ATerm * bottomOfStack)
{
        int lcv;
        static ATbool initialized = ATfalse;
  ATbool help = ATfalse;

        if (initialized)
                return;

        /*{{{  Handle arguments */

        for (lcv=1; lcv < argc; lcv++) {
                if (streq(argv[lcv], SILENT_FLAG)) {
                        silent = ATtrue;
                } else if(streq(argv[lcv], VERBOSE_FLAG)) {
                        silent = ATfalse;
```

File: /home/olivierp/Research/ater... Area: 119,0-119,eol

Figure 4.7: Debugging a stand-alone C program

Figure 4.7 shows the TIDE user interface when debugging a stand-alone program. In this case the program is written in C. A source viewer window shows the source code of the program being debugged. The current point of execution is again highlighted, together with a breakpoint that has been set[3]. Breakpoints are set by double-clicking on the target source line.

To offer the user of TIDE maximum flexibility, we have made it possible to directly edit the event rules present in the debug adapter. In order to do this, the *rule editor* is used, which is shown above the source viewer window in Figure 4.7. In this case the rule that corresponds to the breakpoint highlighted in the source viewer is selected. This direct control over the content of a rule can, for instance, be used to turn an unconditional breakpoint into a conditional one.

4.6 Related work

Reusability of debugger implementations has (at least) two dimensions: portability across multiple platforms and support for multiple languages. Most work on portable debuggers focussed on one of these two dimensions.

Work along the other dimension, achieving some degree of machine independence, mostly aims at supporting only one implementation language across multiple platforms. Probably the best example of this is the work done on ldb [RH92] and more recently cdb [HR96, Han99b], two largely machine independent debuggers for lcc [FH95] a retargetable C compiler.

Cdb indeed nearly achieves complete independence from architectures and operating systems, by loading a small amount of code with the target C program, and by having the compiler emit a machine-independent symbol table.

In some sense, TIDE and cdb could complement each others strengths, as TIDE relies heavily on 'foreign' low level debug implementations. These implementations are often completely machine dependent. Writing a debug adapter based on cdb would provide instant TIDE support across all platforms that cdb runs on.

The extensible graphical debugger Deet [HK97], which is based on cdb, is very closely related to our work. Although it can only be used to debug C programs, it does show how to use gdb as a vehicle to gather low level debugging events. The use of debugging 'nubs' by both cdb and Deet closely resembles the role of debug adapters in our work.

Another example of a related approach is described in [Sos95], where the *Dynascope* program directing tool is described. Dynascope is language independent because it offers a procedural interface for debuggers at an abstraction level *below* that of high-level languages. Debuggers that want to make use of this interface and want to offer high-level language support are forced to implement the mapping between the low abstraction level offered by Dynascope, and the high-level language.

Mainstream debugging technology either totally ignores the issue of reusability (for instance the Microsoft Visual C++ debugger), or supports a small set of different languages on a specific platform (SUN's dbx is a good example).

[3]Normally, a breakpoint would be displayed in red, and the current point of execution would be blue. Unfortunately, this thesis is printed using only black ink.

The only mainstream debugger we know of that supports multiple languages on multiple platforms is the GNU debugger gdb [SPSS00]. The implementation of gdb is also quite monolithic, making the addition of new features very difficult. As a result, the basic set of features offered by gdb has not changed much over the last couple of years.

The effort invested in gdb over the years to get it running on a large number of unix platforms did prompt us to build a debug-adapter on top of it. This debug-adapter yields TIDE support for all the platforms that gdb runs on.

4.7 Conclusions

We have introduced a framework for distributed debugging, by systematically building on well known sequential debugging techniques and implementations. The resulting distributed debugger prototype is unique, both in its simple design and in its flexibility towards the support of new source languages.

One of the strengths of the TIDE approach is that it introduces an extremely clear separation between the platform dependent pieces (debug-adapters) and the platform independent parts. In Table 4.4 the size of the different components of TIDE are shown. The small size of the debug-adapters compared to the platform independent parts is a clear indication of the amount of reuse that can be obtained with the TIDE approach.

The performance of TIDE is comparable to that of traditional sequential debuggers, because primitive events are filtered locally when there is no need for them. Only interesting events are processed in a distributed fashion.

Because of the modularity and simplicity of our framework, we believe it to be a solid base for future experiments. Before starting any new experiments however, we need to add support for more source languages, especially for languages based on paradigms other than the imperative or functional paradigm, for instance logical languages. Our work on debugging of ASF+SDF specifications (see Chapter 8) shows that support for backtracking is straightforward. Extensions in the other direction, more debugging functionality, are also needed. It will be interesting to find out which existing debugging features can be implemented in a generic setting as presented in this chapter.

We also would like to extend the framework to allow *event abstraction* by grouping basic events into combined "abstract events". By visualizing these abstract events [Kun95], we could offer more help in understanding the distributed system being debugged at the application level rather than at the implementation level.

Component	Language	Lines of code
Control center	Java	1643
Java adapter library	Java	1274
C adapter library	C	913
Debug tool library	Java	1704
Total basic TIDE infrastructure		4534
Process-list tool	Java	415
Source viewer	Java	920
Process viewer	Java	894
Rule tracer	Java	677
Rule viewer	Java	472
Animation viewer†	Java	3457
Total debug tools		6844
GDB debug adapter	Java	940
Java debug adapter	Java	1363
ToolBus debug adapter	Java	906
Tcl/Tk debug adapter	C	599
ASF+SDF debug adapter	C	185
Total debug adapters		4448
Total code size of TIDE		15817

†Implemented by H.A. de Jong in the context of his masters thesis [dJ99].

Table 4.4: Size of TIDE components

Part II

Executing and Debugging ASF+SDF Specifications

5 The ASF+SDF Meta-Environment

The old ASF+SDF Meta-Environment has become a legacy system over the last few years. Most of the work in this thesis has been done in the context of developing a new implementation of the ASF+SDF Meta-Environment, which is based on the latest techniques concerning the coupling of software components, construction of user interfaces and programming languages. The second part of this thesis is centered around two of the key techniques integrated in the new ASF+SDF Meta-Environment: the *compilation* and *debugging* of ASF+SDF specifications. We start this second part with a description of this new ASF+SDF Meta-Environment.

The general architecture of the implementation of the new ASF+SDF Meta-Environment is discussed as well as the components which are currently implemented and operational in that environment. Each component is independent of the other components and communicates using the TOOLBUS.

5.1 Introduction

In the beginning of the eighties the design and implementation of the current version of the ASF+SDF Meta-Environment [Kli93] was started. On top of CENTAUR [BCD$^+$89] a programming environment (generator) for writing language definitions in ASF+SDF [HHKR92, BHK89, DHK96] was developed. An overview of these activities can be found in [HK95].

The implementation could be considered a test case for all kinds of ideas concerning the lazy and incremental generation of scanners [HKR92], parsers [HKR90], and term rewriting machines. The development of advanced hybrid editing techniques [Koo94], origin tracking techniques [Deu94], incremental rewriting [Meu94], automatic generation of unparsers [BV96], debugging facilities of term rewriting [Tip91], and the generation of LaTeX code [Vis95] were performed in or with this implementation as well.

The current implementation of the ASF+SDF Meta-Environment has a number of drawbacks and shortcomings, the most important ones are listed below:

- The user interface is outdated and lacks in user-friendliness. Many operations like editing or deleting a module require the user to select a specific module. To do this, the user interface presents all modules in a single list. This can be very cumbersome when working on a large specification (more than 100 modules).

- An often heard complaint is: "The editor is too restricted, why is it not emacs- or vi-like?"
- It is not possible to deploy generated tools independent of the ASF+SDF Meta-Environment.
- It is impossible to port the ASF+SDF Meta-Environment to different architectures. The implementation language (LELISP [LeL90]) is essentially obsolete, and only available on a limited number of platforms.
- Most of the implementation is centered around the tree formalism VTP [Aus90]. This formalism is difficult to use, and the connection between LELISP and VTP is cumbersome.
- New research ideas are hard to implement.
- The current monolithic system is hard to maintain. Bugs are not fixed anymore, because the knowledge about the intrinsics of the system needed to fix these bugs is no longer present.

These points show that the system has all the signs of a *legacy system*, mainly because most of the coding has been done by Ph.D. researchers, and consequently the project has had a large turnover of staff. More detailed lists of complaints and shortcomings together with the requirements for a new ASF+SDF Meta-Environment can be found in [BHK97].

These complaints initiated a redesign and re-implementation of the ASF+SDF Meta-Environment. Initially, it was believed that an incremental re-implementation of the ASF+SDF Meta-Environment was feasible, and therefore a number of people started working on the design and implementation of a new user interface and the replacement of the text editing facilities of GSE [Koo94] by Emacs and Epoch in 1992 [KB93]. However, it proved that is was impossible to manage the interaction between the different tools. This initiated the development of the TOOLBUS software interconnection architecture introduced in Chapter 2. The TOOLBUS will be the backbone of the implementation of the new ASF+SDF Meta-Environment.

Based on the experiences gained with the Epoch-GSE-UI coupling the decision was made to design and implement the new ASF+SDF Meta-Environment from scratch. The fact that the version of LELISP on which the ASF+SDF Meta-Environment was based was becoming obsolete made a "from scratch" approach even more urgent. In this chapter we discuss a first prototype of the new ASF+SDF Meta-Environment based on the TOOLBUS. This prototype offers an extendible infrastructure to experiment with various designs.

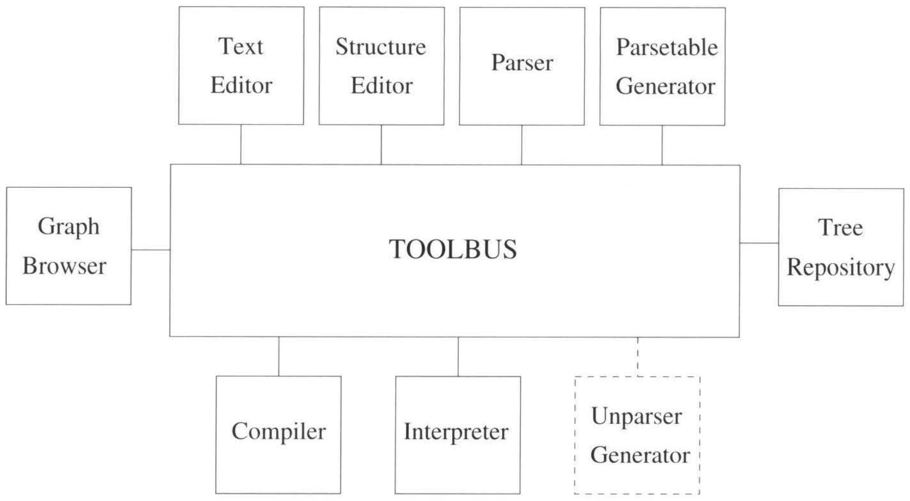


Figure 5.1: General architecture of new ASF+SDF Meta-Environment.

In the rest of this chapter the most important components of the new ASF+SDF Meta-Environment are presented. In Section 5.3 the architecture of the new ASF+SDF Meta-Environment is discussed. Section 5.4 describes the tree-repository to store ASF+SDF modules and terms, furthermore the tree representation format is briefly discussed. The user interface is discussed in Section 5.5, the structure editor in Section 5.6, the parser and parser generator in Section 5.7, the compiler is introduced in Section 5.8 and the interpreter is discussed in Section 5.9.

5.2 Examples of ASF+SDF Specifications

We don't give examples of ASF+SDF specifications here, but refer the interested reader to Figures 7.1, 7.2, 8.1, 8.2, 8.5, and 8.9 in later chapters.

5.3 General Architecture

The architecture of the new ASF+SDF Meta-Environment is depicted in Figure 5.1. This figure is a snapshot of the current state of the prototype, see Table 5.1 for a more detailed list of components and their implementation languages. The unparser generator is currently only available as a stand-alone tool, but will be integrated in the ASF+SDF Meta-Environment in the future.

Table 5.1 gives an overview of all currently available components in the prototype. For

Component	Specification Language	Implementation Language
ATerms	ASF+SDF	C, Java, Tcl, Emacs lisp
AsFix	ASF+SDF	C
TOOLBUS	ASF+SDF	C
User Interface		Tcl/Tk, TclDot
Text editor		Emacs lisp
Structure editor	ASF+SDF	Java
Parse table interp.	ASF+SDF	C
Parse table gener.	ASF+SDF	C
Tree repository	ASF+SDF	C
ASF+SDF Compiler	ASF+SDF	ASF+SDF
Interpreter	ASF+SDF	C
Unparser Generator	ASF+SDF	ASF+SDF

Table 5.1: Components of the new ASF+SDF Meta-Environment

each component it is listed whether this component is specified and in which language it is implemented. The first two components in the table are in fact datatypes, which are used by all other components.

The new ASF+SDF Meta-Environment is based on the TOOLBUS. This means that the TOOLBUS is used as a communication mechanism for the various components available in the environment. The components can not communicate directly with each other. The TOOLBUS script takes care of *all* communication between the components. This script should allow a maximal freedom in order to facilitate future experiments, for instance addition of new components. The information exchange between components is done using the ATerm format (see Section 5.4.1 and Chapter 6 for more details).

5.4 The tree-repository

All components except the debugger manipulate some form of (abstract) syntax trees. It is the responsibility of the tree-repository to store these trees. Before discussing the implementation details of the tree-repository, we discuss the general format of all stored and exchanged information.

5.4.1 Tree Representation

In the old ASF+SDF Meta-Environment the abstract syntax trees are represented by means of VTP (Virtual Tree Processor) [Aus90] offered by CENTAUR [BCD$^+$89]. There are two problems connected to VTP: it is hard to learn programming in VTP, and VTP does not offer enough facilities to prevent illegal access to constructed trees. The latter drawback caused a number of the maintenance problems in the old ASF+SDF Meta-Environment.

These "VTP-problems" led to the development of an alternative formalism to represent syntax trees called AsFix. The AsFix formalism is an instantiation of a generic *annotated term* format: ATerms (see Chapter 6).

ATerms are used to represent structured information to be exchanged between a heterogeneous collection of tools. The ATerm format should be independent of any specific tool or implementation language, but it should be capable of representing *all* data that is exchanged between tools. Consider the following example ATerms:

constant	`abc`
numeral	`123`
literal	`"abc"` and `"123"`
list	`[] [1, "abc", 3]` and `[1, 2, [3,4], 5]`
function	`f("abc")`
annotation	`f(123){color("red"),path([0,2,1])}`

The data format used in the TOOLBUS is also based on ATerms. So all functions for processing, constructing, and accessing ATerms can be used on the TOOLBUS level as well.

These functions have been formally specified in ASF+SDF and we have build library implementations in C and Java that are all based on this formal specification. These libraries are used in all components to manipulate terms.

5.4.2 Representing Syntax Trees: AsFix

The ATerm data type proves to be a powerful and flexible mechanism to represent syntax trees. By defining an appropriate set of function symbols parse trees and abstract syntax trees can be represented for any language or formalism. As an example, we present AsFix, a parse tree format for ASF+SDF.

AsFix The AsFix format (ASF+SDF Fixed format) is an incarnation of ATerms for representing ASF+SDF.

Using AsFix, each module or term is represented by its parse tree which contains both the syntax rules used and all original layout and comments. In this way, the original source text can be reconstructed from the AsFix representation, thus enabling transformation tools to access and transform comments in the source text. Since the AsFix representation is self-contained (all grammar information needed to interpret the term is also included), one can easily develop tools for processing AsFix terms which do not have to consult a common database with grammar information. Examples of such tools are a (structure) editor or a rewrite engine.

AsFix is defined by an appropriate set of function symbols for representing common constructs in a parse tree. These function symbols include the following:

- prod(T) represents production rule T.
- appl(T_1, T_2) represents applying production rule T_1 to the arguments T_2.
- l(T) represents literal T.
- sort(T) represents sort T.
- lex(T_1, T_2) represents (lexical) token T_1 of sort T_2.
- w(T) represents white space T.
- attr(T) represents a single attribute.
- attrs(T) represents a list of attributes.
- no-attrs represents an empty list of attributes.

The following context-free syntax rules (in SDF [HHKR92]) are necessary to parse the input sentence `true or false`.

```
sort Bool
context-free syntax
  true          -> Bool
  false         -> Bool
  Bool or Bool -> Bool {left}
```

The parse tree below gives the AsFix representation for the input sentence `true or false`.

```
appl(prod([sort("Bool"),l("or"),sort("Bool")],sort("Bool"),
          attrs([attr("left")])),
     [appl(prod([l("true")],sort("Bool"),no-attrs),[l("true")]),
      w(" "),l("or"),w(" "),
      appl(prod([l("false")],sort("Bool"),no-attrs),[l("false")])
     ])
```

Two observations can be made about this parse tree. First, this parse tree is an ordinary ATerm, and can be manipulated by all ATerm utilities in a completely generic way.

Second, this parse tree is completely self-contained and does not depend on a separate grammar definition. It is clear that this way of representing parse trees contains much redundant information. Therefore, both maximal sharing and BAF (as presented in Chapter 6) are essential to reduce their size.

The annotations provided by the ATerm data type can be used to store auxiliary information like position information derived by the parser or font and/or color information needed by a (structure) editor. This information is globally available but can be ignored by tools that are not interested in it.

5.4.3 Implementation

The tree-repository contains the AsFix representation of all modules of a specification under construction and of all terms being edited or rewritten. The tree-repository provides functions to add or remove a module or term, and to clear the entire repository. It is possible to check whether a module or term is already in the repository. Furthermore, given the name of a module it is possible to retrieve a specific section of a module, such as its import section or its equations. It is also possible to compute the transitive closure of import relations of a module. The tree-repository is implemented as a table with the module name and term name as key and the AsFix representation as value.

5.4.4 Discussion

The information stored in the tree-repository could be extended with all kinds of extra information such as the size of the file used for the persistent storage of each term, its creation date, etc. Furthermore, the tree-repository should provide a sophisticated querying mechanism as described in [BKV96]. Such a mechanism can be used to locate specific definitions of sorts, lexical and context-free syntax rules, and the like.

5.5 User Interface

Figure 5.2 contains a screendump of the user interface of the current prototype. From left to right one can clearly distinguish the visual representation of the import graph, the list of modules loaded and the buttons to perform actions on modules.

The user interface of the new ASF+SDF Meta-Environment is built around a visual representation of the import graph of a given ASF+SDF specification. The major advantage of having such a visual representation as a basis for the user interface is the increased insight in the structure of the specification. Furthermore, effective visualization of this graph can reveal interesting characteristics of the specification (e.g. repeating patterns and unintended transitive import relations).

The user can select one or more modules in this graph and perform actions on them. Currently, the following actions are supported:

- Open the module editor for this module.
- Open a term editor over this module.
- Delete the module from the repository.
- Request additional information about the module.
- Revert the module in the repositories to the last version saved to disk.

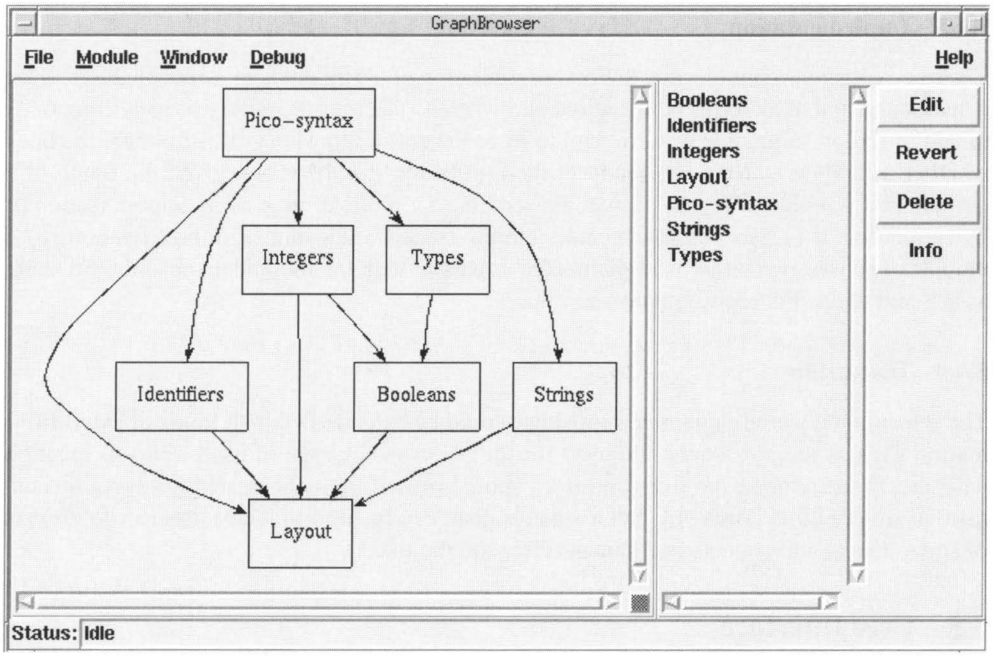


Figure 5.2: Browsing the import graph in the new ASF+SDF Meta-Environment.

These actions can be selected in a menu which pops up when the user clicks on a module in the graph, or using the module list and buttons on the right-hand-side of the screen. In addition to the actions described above, the user can:

- Add a module to the specification loaded in the repositories.
- Clear all repositories.
- Revert all modules loaded in the repositories to the last versions saved to disk.
- Save the import graph in various graphics formats.

5.5.1 Implementation

The user interface is implemented in Tcl/Tk [Ous94] and TclDot: an extension for Tcl that incorporates the directed graph facilities of *dot* [KN93] into Tcl and provides a set of commands to control those facilities. Basically, TclDot contains commands to define a graph, add nodes and edges to this graph, and compute the placement of nodes and edges of this graph. The layout is computed in a way that tries to expose the logical structure of the graph, avoid crossings of edges, keep the edges short and emphasize symmetry and balance [GKNV93]. TclDot is part of the *graphviz* package [NK94, EKN96]: a set of graph drawing tools for Unix or MS-Windows from AT&T/Lucent Technologies.

The user interface sends events to the TOOLBUS to add, delete, revert, or edit a module, to edit a term over a module, to display extra information about a module, or to quit the ASF+SDF Meta-Environment. These events are handled by the TOOLBUS-script which may distribute them to other tools for further processing.

The user interface is notified by the TOOLBUS of state changes in the list of modules, or in the import relations of a module so it can updates its visual representation of the import graph.

5.5.2 Future Work

Future work on the user interface includes a mechanism for searching in modules (in cooperation with the tree-repository and the editor). Using this mechanism it should for example be possible to highlight/select all modules that use a given function or sort.

Furthermore, the current version of TclDot only supports static graphs. This means that the layout of a graph is computed from scratch every time an update is performed (i.e. adding or removing a node or edge). The result of this computation can be completely different from the original graph. This is undesirable for a user interface since it can be confusing and the user needs to familiarize himself with a new structure. A new version of TclDot, called TclDG, supports so called *dynamic graphs* [Nor96, EN96]. The layout of these graphs changes incrementally when updates are performed. This results in more gradual changes in the structure of the graph. As soon as the TclDG package is released, it should be incorporated in the user interface.

Finally, it is convenient if the user can adapt the layout of the import graph to clarify its logical structure (e.g., move nodes/edges to improve their ordering). These edit operations should cooperate in some way with the layout mechanism so that changes of the user are not undone by layouting the graph. An example of a more rigorous editing operation is the ability to define subgraphs in the import graph which can be collapsed into a single node. Such a feature can be useful to improve the readability of big import graphs.

5.6 Editors

The structure editing system in the new implementation provides roughly the same functionality as the Generic Structure Editor (GSE [Koo94]). There are, however a number of differences, both in the external and the internal behavior.

5.6.1 Internal Behavior

The structure editing system consists of two parts. One is a *structure* editor, the other is a *text* editor. The structure editor operates on parse trees (encoded as AsFix terms). It only manipulates (sub)trees, i.e., it does not manipulate the lexical content of nodes in a parse tree. The text editor operates on a character level, it *does* manipulate the content of nodes in a parse tree.

Both the text editor and the structure editor have a well defined external behavior (a ToolBus interface) [Kui96]. This makes it possible to use any (existing) text editor as long as it adheres to the interface. One of the main weaknesses of GSE has always been its limited text editing capabilities. By separating text and structure editing functionality we hope to address this problem.

The text and structure editors are tied together by means of a ToolBus script. The script provides us with a one-to-one mapping from text to structure and back. It makes sure that at any given time the data structure in the text editor (the text) can be translated to the data structure in the structure editor (the parse tree), and vice versa. If a string α is a syntactically correct string in the language L, and we have a parser Π_L over this language and a pretty printer pp then $pp(\Pi_L(\alpha)) = \alpha$. As a consequence, if t is the parse tree that results from $\Pi_L(\alpha)$ then also $\Pi_L(pp(t)) = t$.

5.6.2 External Behavior

If the edited text is not syntactically correct (which is inevitable during editing) then the smallest subtree that contains an incorrect program fragment will be held in a focus. In GSE a similar approach is used. The main difference between GSE and the new structure editor is that GSE has two specific modes, one for text editing and one for structure editing. Once the switch to text-editing mode is made, all structure information is lost.

The new editor does not need this distinction. It allows text-editing while retaining structure information. Furthermore, the new structure editor can create multiple focuses, thus minimizing the amount of text that needs to be (re)parsed after an edit session.

The difference between these approaches is perhaps best illustrated with an example.

Consider the following program in the language While:

```
x := 10 ;
while x > 0 do x := x - 1
```

Now suppose we want to decrease x by 2 during each iteration. We replace the character `1` by the character `2`. After this replacement the focus will be on the integer 2 (The underlined character).

```
x := 10 ;
while x > 0 do x := x - 2
```

Now suppose we want to edit the stop condition of the while loop, such that the loop terminates when x is greater than 2. In GSE the focus would then look like this:

```
x := 10 ;
while x > 2 do x := x - 2
```

In the new editor, instead of increasing the focus, a new focus will be created, which looks like this:

```
x := 10 ;
while x > 2 do x := x - 2
```

One of the motivations for using structure editors is the fact that they allow us to parse text incrementally. Only the parts of the text that have been changed need to be reparsed. As shown in the last example: this strategy obviously results in less parsing than in GSE.

However, this is not always the case. If we take the original program again, and decide to put the body of the while loop between brackets, we get the following focus positions.

```
x := 10 ;
while x > 0 do {x := x - 1}
```

where in GSE we would have had

```
x := 10 ;
while x > 0 do {x := x - 1}
```

In this case, the last solution is better, because the first solution leaves us with two syntactically incorrect focuses. However, there are a number of strategies that could help us here. In this case, the solution would be to create a new focus that exactly contains all the old focuses, effectively giving the same functionality as GSE.

5.6.3 Implementation

As stated above, the editing system has been implemented as two separate tools. The structure editor was first specified in ASF+SDF, and after prototyping it in Java it was finally implemented in C for performance reasons. The text editor is currently based on emacs, and all ASF+SDF Meta-Environment specific functionality is coded in emacs lisp. The interface of the text editor is flexible enough to add support for other editors in the future.

The parsing strategies mentioned in the previous section are implemented as part of the TOOLBUS-script. In the script we decide whether a focus should be parsed, or whether it should first be expanded, and then parsed.

5.6.4 Future Work

In Section 5.6.2 we mentioned the need for experimenting with different parsing strategies. By moving the implementation of these strategies out of the structure editor and into the TOOLBUS-script, we hope to create a system that provides us with the flexibility to try out different strategies.

Last but not least, we need to realize a link between the structure editing system and the tree-repository (Section 5.4). In GSE, when expanding a meta-variable, the editor lists all productions with the same sort as the meta-variable. As the new editing system is completely language independent, it needs to get the list of productions from the tree-repository explicitly.

5.7 Parsing

The user definable syntax of ASF+SDF makes the parsing technology in the new ASF+SDF Meta-Environment very important. We need a parser that can be instantiated with each language over which we want to parse terms. To do this, we have a *parser generator* that generates a parse table given the syntax of a language. These parse tables are used to instantiate the parser (SGLR) to parse terms over the syntax.

5.7.1 Parser Generator

From an SDF syntax definition a parse table is generated for later use by SGLR (see Section 5.7.2). Note that the new parser generator is no longer based on incremental and lazy generation techniques [HKR90]. The generation process consists of two distinct phases. In the first phase, the SDF definition is normalized to an intermediate, rudimentary, formalism: *Kernel-SDF*. In the second phase this Kernel-SDF is transformed to a parse table.

Grammar Normalization The grammar normalization phase consists of several steps and concludes by producing a Kernel-SDF definition. The most important steps are the following:

- A modular SDF specification is transformed into a flat specification.
- Lexical grammar rules are transformed to context-free grammar rules.
- Priority and associativity definitions are transformed to a list of pairs, where each pair consists of two production rules for which a priority or associativity relation holds. The transitive closure of the priority relations between grammar rules is made explicit in these pairs.

Parse Table Generation The actual parse table is derived from the Kernel-SDF definition. To do so, a rather straightforward SLR(1) approach is taken. However, possible shift/reduce or reduce/reduce conflicts are unproblematic, and can hence simply be stored in the table. Some extra calculations are then performed in order to reduce the number of conflicts in the parse table. Based on the list of priority relation pairs the table is filtered; see [KV94] for more details. The resulting table contains a list of all Kernel-SDF production rules, a list of states with the actions and gotos, and a list of all priority relation pairs. The parse table is represented as an ordinary ATerm, see Chapter 6.

5.7.2 Scannerless Generalized LR Parsing (SGLR)

Even though parsing is often considered a solved problem in computer science, every now and then new ideas and combinations of existing techniques pop up. SGLR (Scannerless Generalized LR) parsing is a striking example of a combination of existing techniques that results in a remarkably powerful parser.

Generalized LR *Parsing for Context-Free Grammars* LR parsing [ASU86] is a well-known parsing technique used in many well-known implementations, e.g. LEX/YACC [LS86, Joh86] and FLEX/BISON (the GNU counterpart of LEX/YACC). LR parsers are based on the shift/reduce principle; a (conflict-free) LR(k) ($k \geq 0$) parse table, containing actions and gotos, is used. A conventional LR parser consists of a scanner, that splits the input stream into tokens, and a parser that processes the tokens and either generates error messages or builds a parse tree.

LR parsing restricts the class of languages that can be recognized to LR(k) ($k \geq 0$). Although all kinds of subtleties exist (such as LALR(k)), these are of no relevance to this discussion.

The ability to cope with arbitrary context-free grammars is of major importance if one wishes to allow a modular syntax definition formalism. Due to the fact that LR(k)-grammars are not closed under union, a more powerful parsing technique is required. Generalized LR-parsing [Tom85, Rek92] (GLR-parsing) is a natural extension to LR-parsing, from this perspective. The saving grace of GLR-parsing lies in the fact that it does not require the parse table to be conflict-free. Allowing conflicts to occur in parse tables, GLR is equipped to deal with arbitrary context-free grammars.

If, while parsing, a conflict in the parse table occurs, the parse stack is split; every possible continuation gets its own parse stack. These stacks are processed in parallel, but the stacks are synchronized when shifting input tokens. Stacks sharing the same state are merged at that point.

The parse result, then, might not consist of a single parse tree; generally, a forest consisting of an arbitrary number of parse trees is yielded. Ambiguity produces multiple parse trees, each of which embodies a parse alternative. In case of an LR(1) grammar, the GLR algorithm collapses into LR(1), and exhibits similar performance characteristics. As a rule of thumb, the simpler the grammar, the closer GLR performance will be to LR(1) performance.

Eliminating the Scanner The GLR parser as it is found in the old ASF+SDF Meta-Environment uses a scanner, just like any conventional LR(k) parser. The use of a scanner in combination with GLR parsing leads to a certain tension between scanning and parsing. In a number of situations the scanner has several ways of recognizing a string of characters: there is a so-called lexical ambiguity[1]. At that point, the scanner has to come to some decision; later, when parsing the tokens as output by the scanner, the selected tokenization might turn out not to be what the parser expected, causing the parse to fail.

[1]Consider the well-known examples of the range operator vs. the real numbers in Pascal.

Scannerless GLR parsing [Vis97] solves this problem by unifying scanner and parser. In other words: the scanner is eliminated by simply considering all elementary input symbols to be input tokens for the parser. Each character becomes a separate token, and ambiguities on the lexical level are dealt with by the GLR algorithm. To accomplish this level of integration, the lexical syntax rules are transformed into context-free syntax rules.

5.7.3 Implementation

The parse table generator The parser generator consists of two phases as described in Section 5.7.1. Both phases where initially specified in ASF+SDF, and compiled to C. This method of implementation proved too inefficient for the second phase, the generation of the actual parse table. Consequently, we have reimplemented this phase in C. A good illustration of the performance gained this way is the time it now takes to generate the parse table for COBOL. Before the reimplementation this generation would take about 78 minutes. Using the C implementation this time was reduced to about 80 seconds. Only about 10 of these 80 seconds are used by the second phase. It is clear that any new gains must come from the first phase, for instance by also reimplementing it in C.

The parser For efficiency reasons, SGLR is implemented in C. Via the TOOLBUS, SGLR can be instantiated with parse tables. After this instantiation, strings can be parsed using one of these parse tables. The result can be one of three things:

- A parse tree after a successful parse without ambiguities.
- A parse forest after a successful parse where ambiguities were encountered.
- A parse error when the input string did not form a valid sentence over the input syntax.
- A cycle detection error when the input syntax contained cycles. Not all of these cycles can be detected statically by looking at the input syntax, so cycle detection errors can be encountered when parsing.

5.7.4 Discussion

The current parser is not very good in error diagnosis, and error recovery is lacking completely. An elegant solution for handling ambiguities is also needed.

A second problem with the current implementation is related to the modular nature of ASF+SDF specifications. Because the syntax definitions are also modular, a specification actually consists of a hierarchical structure of language specifications. In the current situation, for each module a different parse table is needed that represents the syntax of that module and all of its imported modules.

Clearly, each of these parse tables has a lot in common with the parse tables of its imported modules. More research is needed to add some kind of modularization support to both the parse table generator and the parser, in order to reduce this redundancy.

5.8 Compiler

The new ASF+SDF Meta-Environment provides two mechanisms for executing specifications. The interpreter is used for incremental development of specifications, and is discussed in the next section. The performance of the interpreter is not good enough for industrial strength applications, but the turnaround time for changes in specifications is excellent. When a specification is finished, the ASF+SDF compiler can be used to generate C code that efficiently rewrites terms according to the specification.

The ASF+SDF compiler will be discussed in full detail in Chapter 7.

5.9 Interpreter

The interpreter or evaluator takes care of rewriting terms given a set of equations. The interpreter rewrites terms in AsFix format using equations in AsFix format. A first version of the interpreter was specified in ASF+SDF. Based on this specification an efficient version was handcrafted in C.

We will first discuss how the interpreter is activated and which steps are performed before a term is actually rewritten. Then we will discuss the implementation of the interpreter in more detail. Finally, we will discuss some related aspects, such as performance, improvements, etc.

5.9.1 Activating the Interpreter

The interpreter is activated in the same way as in the old ASF+SDF Meta-Environment, each term editor is extended with a `Reduce`-button. After pushing this button a check is performed whether the interpreter has the appropriate set of rewrite rules available. If not, all equations of the modules in the transitive closure of the import graph are retrieved from the module repository.

When the interpreter receives a set of equations it performs some simple transformations on it, for instance, layout is removed and lexicals are transformed into lists of characters. The transformation is performed in order to use the standard list matching mechanism to deal with lexicals. The interpreter stores the equations in a hash table to have fast access to them during rewriting.

The term to be rewritten is also slightly modified, layout is removed and the lexicals are transformed into lists of characters. After rewriting, the result term is again modified: layout is inserted and the lists of characters are translated back into lexicals. The inserted layout is rather arbitrary, to get a better layout of the reduced term it is necessary to adapt this standard unparsing mechanism, see [BV96] for more details. This term is sent to the TOOLBUS to be displayed in an editor.

The interpreter does not throw away the equations after rewriting. The equations are only thrown away when one of the modules in the specification is modified.

5.9.2 Implementation

Before discussing the implementation of the interpreter we recall some of the characteristics of the ASF+SDF-formalism, and more specific of ASF itself. The ASF-formalism has the following characteristics:

- The functions are many-sorted.
- The equations may be non left-linear.
- It is possible to use list matching.
- The conditions in the equations may be both positive and negative.
- It is possible to use default equations.
- The evaluation strategy of the equations is based on innermost rewriting.

The main functionality of the interpreter consists of a rewriting machine and a local repository to store equations. This repository is organised as a table, the keys in this table are module names and the values are sets of equations corresponding to the transitive closure of the import graph of the corresponding module. The C implementation of the shared ATerm library takes care of unnecessary duplication of the rewrite rules. In the C implementation of the interpreter the set of equations is stored in a hash-table and the hash key is calculated using the outermost function symbol of the left hand side of an equation and the outermost function symbol (if any) of the first argument of the left hand side of the equation. This improves the efficiency of the rewriting machine enormously, but it influences the semantics as well. In fact this implements a form of syntactic specificity, because when the interpreter is looking for an equation that matches with a term, it first looks for an equation that has the same outermost function symbol (OFS), *and* has the same OFS at the first argument position. If no match can be found, the search is continued for an equation that has the same OFS as the term, but with a variable at the first argument position. This strategy means that equations with a variable at the first argument position are only applied when no other equation is applicable. Finally, when this search also fails the interpreter looks for a default equation that matches with the term.

The rewrite machine itself consists of a collection of recursive functions which have as arguments a set of equations, the term to be rewritten, and an environment in which the instantiated variables are stored.

Recursion is used to implement the backtracking behavior of list matching in ASF. The instantiation of list variables is done by assigning an "arbitrary" sublist to a list variable. If this does not lead to a successful matching of all variables in the equation or one of the conditions can not be satisfied, another sublist is tried. This process is repeated until either a successful match is found, or all sublists are tried.

ASF+SDF allows the use of conditional equations. The conditions may be positive as well as negative. The current prototype does not allow the introduction of new variables in a negative condition. Furthermore, it is not allowed to introduce new variables on both sides of a positive condition. If new variables are introduced on one side of a positive condition only the other side is rewritten which is then matched against the "variable introducing side" of the condition, leading to new variable bindings.

5.9.3 Discussion

There are a few open issues with respect to the interpreter. First, although the performance of the current version is reasonable, we feel it can still be improved. When no list matching is involved, the current version is about twice as fast as the old ASF+SDF Meta-Environment. However, performance decreases drastically when matching lists.

When looking for ways to improve the performance of the interpreter, it is important to keep in mind that it will also be used as a tool to experiment with aspects of the execution of ASF+SDF specifications. This means that the implementation must be kept open and easy to comprehend.

One of the areas in which we are currently experimenting is debugging of ASF+SDF specifications. In Chapter 8 we show how the interpreter can be connected to the TIDE system described in Chapter 4.

It will indeed prove challenging to keep improving the performance of the interpreter without sacrificing extensibility.

One technique that will allow us to significantly gain performance, is the use of preprocessing of specifications. A number of preprocessing steps could improve the performance considerably. One obvious preprocessing step is the calculation of which side of a positive condition introduces new variables. Another very effective preprocessing step is the transformation of some forms of list matching into non list matching, e.g., obtaining the head and tail of a list, etc. This can be done because we know the internal data structure for lists in the interpreter. These kinds of list transformations are also important when compiling ASF+SDF specifications (see Chapter 7).

5.10 Debugging issues

Debugging in the context of the new ASF+SDF Meta-Environment has three dimensions:

1. Debugging of the new ASF+SDF Meta-Environment itself
2. Debugging of ASF+SDF specifications
3. Debugging of programs written in languages specified in ASF+SDF

Our work presented in the first part of this thesis can be used to cover all three dimensions.

The ASF+SDF Meta-Environment is a sophisticated distributed system that is based on the TOOLBUS. The TIDE system discussed in Chapter 4 can therefore be used directly to debug the ASF+SDF Meta-Environment. The individual components can be debugged as long as they are executed using a language implementation that is supported by TIDE. Luckily, this is the case for almost all components in the ASF+SDF Meta-Environment, as the implementation languages are C, Java, Tcl/Tk, and ASF+SDF. Only the text editor, which is written in emacs lisp, cannot be debugged using TIDE at this time.

The latter two dimensions, debugging of ASF+SDF specifications and debugging of programs written in languages that are specified in ASF+SDF, are discussed in Chapter 8.

5.11 Conclusions

In this chapter the first prototype of the new ASF+SDF Meta-Environment is discussed. This version of the prototype should be considered as a test case to see whether for instance the TOOLBUS is suited as backbone for the new ASF+SDF Meta-Environment. One of the lessons we learned from the implementation of the old ASF+SDF Meta-Environment is that it is essential to have a flexible and extendible implementation. The ASF+SDF Meta-Environment is first of all a research tool, which means that it should facilitate the testing of all kinds of new ideas.

6
ATerms

All data exchange between components in the new ASF+SDF Meta-Environment is done using a common data exchange format called ATerms. Many of these components also operate on ATerms internally. Especially because all components that are generated from ASF+SDF specifications use ATerms as the sole datatype on which all computations are based (see Chapter 7). This means that our implementation of the ATerm datatype should not only be able to exchange ATerms quickly, but it should also allow for space and time efficient calculations using ATerms as the main datatype.

To accomplish this, we have designed and implemented the ATerm library described in this chapter. We succeeded in making the operations offered by this library space efficient by using maximal sharing, and we have made it time efficient by carefully selecting implementation techniques like mark-and-sweep garbage collection to reclaim unused terms, and fast hashing techniques to maintain maximal sharing. Efficient exchange of ATerms is accomplished using the binary aterm format (BAF) which is also described in this chapter.

6.1 Introduction

Cut and paste operations on complex data structures are standard in most desktop software environments: one can easily clip a part of a spreadsheet and paste it into a text document. The exchange of complex data is also common in distributed applications: complex queries, transaction records, and more complex data are exchanged between different parts of a distributed application. Compilers and programming environments consist of tools such as editors, parsers, optimizers, and code generators that exchange syntax trees, intermediate code, and the like.

How is this exchange of complex data structures between applications achieved? One solution is Microsoft's Object Linking and Embedding (OLE) [Cha96]. This is a platform-specific, proprietary, set of primitives to construct Windows applications. Another, language-specific, solution is to use Java's serialization interface [GJS96]. This allows writing and reading Java objects as sequential byte streams. Yet another solution is to use OMG's Interface Definition Language (part of the Common Object Broker Architecture [OMG97]) to define data structures in a language-neutral way. Specific language-bindings provide the mapping from IDL data structures to language-specific data structures.

All these solutions have their merits but do not really qualify when looking for an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of complex data structures between distributed applications. To be more specific, we are interested in a solution with the following characteristics:

Open: independent of any specific hardware or software platform.

Simple: the procedural interface should contain 10 rather than 100 functions.

Efficient: operations on data structures should be fast.

Concise: inside an application the storage of data structures should be as small as possible by using compact representations and by exploiting sharing. Between applications the transmission of data structures should be fast by using a compressed representation with fast encoding and decoding. Transmission should preserve any sharing of in-memory representation in the data structures.

Language-independent: data structures can be created and manipulated in any suitable programming language.

Annotations: applications can transparently extend the main data structures with annotations of their own to represent non-structural information.

In this chapter we describe the data type of *Annotated Terms*, or just *ATerms*, that have the above characteristics. They form a solution for our implementation needs in the areas of interactive programming environments [Kli93, BKMO97] and distributed applications [BK98] but are more widely applicable. Typically, we want to exchange and process tree-like data structures such as parse trees, abstract syntax trees, parse tables, generated code, and formatted source texts. The applications involved include parsers, type checkers, compilers, formatters, syntax-directed editors, and user-interfaces written in a variety of languages. Typically, a parser may add annotations to nodes in the tree describing the coordinates of their corresponding source text and a formatter may add font or color information to be used by an editor when displaying the textual representation of the tree.

The ATerm data type has been designed to represent such tree-like data structures and it is therefore very natural to use ATerms both for the internal representation of data inside an application and for the exchange of information between applications. Besides function applications that are needed to represent the basic tree structure, a small number of other primitives are provided to make the ATerm data type more generally applicable. These include integer constants, real number constants, binary large data objects ("blobs"), lists of ATerms, and placeholders to represent typed gaps in ATerms. Using the comprehensive set of primitives and operations on ATerms, it is possible to perform operations on an ATerm received from another application without first converting it to an application-specific representation.

First, we will give a quick overview of ATerms (Section 6.2). Next, we discuss implementation issues (Section 6.3) and give some insight in performance issues (Section 6.4). An overview of applications (Section 6.5) and an overview of related work and a discussion (Section 6.6) conclude this chapter.

6.2 ATerms at a Glance

We now describe the constructors of the ATerm data type (Section 6.2.1) and the operations defined on it (Section 6.2.2).

6.2.1 The ATerm Data Type

The data type of ATerms (`ATerm`) is defined as follows:

- `INT`: An integer constant (32-bits integer) is an ATerm.[1]
- `REAL`: A real constant (64-bits real) is an ATerm.
- `APPL`: A function application consisting of a function symbol and zero or more ATerms (arguments) is an ATerm. The number of arguments of the function is called the *arity* of the function.
- `LIST`: A list of zero or more ATerms is an ATerm.
- `PLACEHOLDER`: A placeholder term containing an ATerm representing the type of the placeholder is an ATerm.
- `BLOB`: A "blob" (Binary Large data OBject) containing a length indication and a byte array of arbitrary (possibly very large) binary data is an ATerm.
- A list of ATerm pairs can optionally be associated with every ATerm representing a list of $(label, annotation)$ pairs.

Each of these constructs except the last one (i.e., `INT`, `REAL`, `APPL`, `LIST`, `PLACE-HOLDER`, and `BLOB`) form subtypes of the data type `ATerm`. These subtypes are needed when determining the type of an arbitrary ATerm. Depending on the actual implementation language they will be represented as a constant (C, Pascal) or a subclass (C++, Java).

The last construct is the *annotation construct*, which makes it possible to annotate terms with transparent information[2].

[1] We have upgraded the ATerm library to support 64-bit architectures as well.

[2] Transparent in the sense that the result of most operations is independent of the annotations. This makes it easy to completely ignore annotations. Examples of the use of annotations include annotating parse trees with positional or typesetting information, and annotating abstract syntax trees with the results of type checking.

Appendix D contains a definition of the concrete syntax of ATerms. The primary reason for having a concrete syntax is to be able to exchange ATerms in a human-readable form. In Section 6.3 we also discuss a compact binary format for the exchange of ATerms in a format that is only suitable for processing by machine. We will now give a number of examples to show some of the features of the textual representation of ATerms.

- Integer and real constants are written conventionally: `1`, `3.14`, and `-0.7E34` are all valid ATerms.
- Function applications are represented by a function name followed by an opening parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: `f(a,b)` and `"test!"(1,2.1,"Hello world!")`. These examples show that double quotes can be used to delimit function names that are not identifiers.
- Lists are represented by an opening square bracket, a number of list elements that are separated by commas and a closing square bracket: `[1,2,"abc"]`, `[]`, and `[f,g([1,2]),x]` are examples.
- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are `<int>`, `<[3]>`, and `<f(<x>,<real>)>`.
- Blobs do not have a concrete syntax because their human-readable form depends on the actual blob content.

6.2.2 Operations on ATerms

The operations on ATerms fall into three categories: making and matching ATerms (Section 6.2.2), reading and writing ATerms (Section 6.2.2), and manipulating ATerms annotations (Section 6.2.2). The total of only 13 functions provides enough functionality for most users to build simple applications with ATerms. We refer to this interface as the *level one* interface of the ATerm data type.

To accommodate "power" users of ATerms we also provide a *level two* interface, which contains a more sophisticated set of data types and functions. It is typically used in generated C code that calls ATerm primitives, or in efficiency-critical applications. These extensions are useful only when more control over the underlying implementation is needed or in situations where some operations that can be implemented using level one constructs can be expressed more concisely and implemented more efficiently using level two constructs. The level two interface is a strict superset of the level one interface (see Appendix E for further details).

Observe that ATerms are a purely functional data type and that no destructive updates are possible, see Section 6.3.2 for more details.

Making and Matching ATerms The simplicity of the level one interface is achieved by the *make-and-match* paradigm:

- *make* (compose) a new ATerm by providing a pattern for it and filling in the holes in the pattern.
- *match* (decompose) an existing ATerm by comparing it with a pattern and decompose it according to this pattern.

Patterns are just ATerms containing placeholders. These placeholders determine the places where ATerms must be substituted or matched. A typical example of a pattern is a term like `"and(<int>,<appl>)"`. These patterns appear as string argument of both make and match and are remotely comparable to the format strings in the `printf/scanf` functions in C. The operations for making and matching ATerms are:

- `ATerm ATmake(String` p`, ATerm` a_1`, ..., ATerm` a_n`)`: Create a new term by taking the string pattern p, parsing it as an ATerm and filling the placeholders in the resulting term with values taken from a_1 through a_n. If the parse fails, a message is printed and the program is aborted. The types of the arguments depend on the specific placeholders used in *pattern*. For instance, when the placeholder `<int>` is used an integer is expected as argument and a new integer ATerm is constructed.
- `ATbool ATmatch(ATerm` t`, String` p`, ATerm *`a_1`, ..., ATerm *`a_n`)`:

  Match term t against pattern p, and bind subterms that match with placeholders in p with the result variables a_1 through a_n. Again, the type of the result variables depend on the placeholders used. If the parse of pattern p fails, a message is printed and the program is aborted. If the term itself contains placeholders these may occur in the resulting substitutions. The function returns `true` when the match succeeds, `false` otherwise.
- `Boolean ATisEqual(ATerm` t_1`, ATerm` t_2`)`: Check whether two ATerms are equal. The annotations of t_1 and t_2 must be equal as well.
- `Integer ATgetType(ATerm` t`)`: Retrieves the type of an ATerm. This operation returns one of the subtypes mentioned before in Section 6.2.1.

Reading and Writing ATerms For reasons of efficiency and conciseness, reading and writing can take place in two forms: text and binary. The text format uses the textual representation discussed earlier in Section 6.2.1 and Appendix D. This format is human-readable, space-inefficient[3], and any sharing of the in-memory representation of terms is lost.

[3]The unnecessary size explosion could be avoided by extending the textual representation with a mechanism for labeling and referring to terms. Instead of `f(g(a),g(a))`, one could then write `f(1:g(a), #1)`. The first occurrence of `g(a)` is labeled with "`1`", and the second occurrence refers to this label ("`#1`").

The binary format (Binary ATerm Format, see Section 6.3.5) is portable, machine-readable, very compact, and preserves all in-memory sharing. The operations for reading and writing ATerms are:

- `ATerm ATreadFromString(String` s`)`: Creates a new term by parsing the string s. When a parse error occurs, a message is printed, and a special error value is returned.
- `ATerm ATreadFromTextFile(File` f`)`: Creates a new term by parsing the data from file f. Again, parse errors result in a message being printed and an error value being returned.
- `ATerm ATreadFromBinaryFile(File` f`)`: Creates a new term by reading a binary representation from file f.
- `Boolean ATwriteToTextFile(ATerm` t`, File` f`)`: Write the text representation of term t to file f. Returns `true` for success and `false` for failure.
- `Boolean ATwriteToBinaryFile(ATerm` t`, File` f`)`: Write a binary representation of term t to file f. Returns `true` for success, and `false` for failure.
- `String ATwriteToString(ATerm` t`)`: Return the text representation of term t as a string.

Either format (textual or binary) can be used on any linear stream, including files, sockets, pipes, etc.

Annotating ATerms Annotations are $(label, annotation)$ pairs that may be attached to an ATerm. Recall that ATerms are a completely functional data type and that no destructive updates are possible. This is evident in the following operations for manipulating annotations:

- `ATerm ATsetAnnotation(ATerm` t`, ATerm` l`, ATerm` a`)`: Return a copy of term t in which the annotation labeled with l has been changed into a. If t does not have an annotation with the specified label, it is added.
- `ATerm ATgetAnnotation(ATerm` t`, ATerm` l`)`: Retrieve the annotation labeled with l from term t. If t does not have an annotation with the specified label, a special error value is returned.
- `ATerm ATremoveAnnotation(ATerm` t`, ATerm` l`)`: Return a copy of term t from which the annotation labeled with l has been removed. If t does not have an annotation with the specified label, it is returned unchanged.

6.3 Implementation

6.3.1 Requirements

In Section 6.1 we have already mentioned our main requirements: openness, simplicity, efficiency, conciseness, language-independence, and capable of dealing with annotations. There are a number of other issues to consider that have a great impact on the implementation, and that make this a fairly unique problem:

- By providing automatic garbage collection ATerm users do not need to deallocate ATerm objects explicitly. This is safe and simple (for the user).
- The expected lifetime of terms in most applications is very short. This means that garbage collection must be fast and should touch a minimal amount of memory locations to improve caching and paging performance.
- The total memory requirements of an application cannot be estimated in advance. It must be possible to allocate more memory incrementally.
- Most applications exhibit a high level of redundancy in the terms being processed. Large terms often have a significant number of identical subterms. Intuitively this can be explained from the fact that most applications process terms with a fixed signature and a limited tree depth. When the amount of terms that is being processed increases, it is plausible that the similarity between terms also increases.
- In typical applications less than 0.1 percent of all terms have an arity higher than 5.
- Many applications will use annotations only sparingly. The implementation should not impose a penalty on applications that do not use them.
- In order to have a portable yet efficient implementation, the implementation language will be C. This poses some special requirements on the garbage collection strategy[4].

With these considerations in mind, we will now discuss maximal (in-memory) sharing of terms (Section 6.3.2), garbage collection (Section 6.3.3), the encoding of terms (Section 6.3.4), and the Binary ATerm Format (Section 6.3.5).

6.3.2 Maximal Sharing

Our strategy to minimize memory usage is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term

[4]We have implemented the library in Java as well. In this case, many of the issues we discuss in this chapter are irrelevant, either because we can use built-in features of Java (garbage collection), or because we just cannot express these low level concerns in Java.

is reused, ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be built and leads to maximal sharing of subterms. The library functions that construct terms make sure that shared terms are returned whenever possible. The sharing of terms is thus invisible to the library user.

The Effects of Maximal Sharing Maximal sharing of terms can only be maintained when we check at every term creation whether a particular term already exists or not. This check implies a search through all existing terms but must be fast in order not to impose an unacceptable penalty on term creation. Using a hash function that depends on the internal code of the function symbol and the addresses of its arguments, we can quickly search for a function application before creating it. The terms are stored in a hash table. The hash table does not contain the terms themselves, but pointers to the terms. This provides a flexible mechanism of resizing the table and ensures that all entries in the table are of equal size. Hence the (modest but not negligible) cost at term creation time is one hash table lookup.

Fortunately, we get two returns on this investment. First, the considerably reduced memory usage also leads to reduced execution time. Second, we gain substantially as the equality check on terms (`ATisEqual`) becomes very cheap: it reduces from an operation that is linear in the number of subterms to be compared to a constant operation (pointer equality).

Another consequence of our approach is less fortunate. Because terms can be shared without their creator knowing it, terms cannot be modified without creating unwanted side-effects. This means that terms effectively become *immutable* after creation. Destructive updates on maximally shared terms are not allowed. Especially in list operations, the fact that ATerms are immutable can be expensive. It is often the responsibility of the user of the library to choose algorithms that minimize the effect of this shortcoming.

Searching for Shared Subterms Maximal sharing of terms requires checking at term creation time whether this term already exists. This search must be fast in order to ensure efficient term creation. A hash function based on the *addresses* of the function symbol and the arguments of a function application allows for a quick lookup in the hash table to find a function application before creating it.

Collisions One issue in hash techniques is handling collisions. The simplest technique is linear chaining [Knu73]. This requires one pointer in each object for hash chaining, which in our implementation implies a memory overhead of about 25 percent. Other solutions for collision resolution will either increase the memory requirements, or the time needed for insertions or deletions (see [Knu73]). We therefore use linear hash chaining in our implementation.

Direct or Indirect Hashing Another issue is whether to store all terms directly in the hash table, or only references. Storing the objects directly in the hash table saves a memory access when retrieving a term as well as the space needed to store the reference. However, there are severe drawbacks to this approach:

- We cannot rehash old terms because rehashing means that we have to move the objects in memory. When using C as an implementation language, moving objects in memory is not allowed because we can only determine a conservative root set and therefore are not allowed to change the pointers to roots. This would mean that the hash table could not grow beyond its initial size.
- Internal fragmentation is increased, because empty slots in the hash table are as large as the object instead of only one machine word.
- We would need a separate hash table for each term size to decrease the internal fragmentation.

Because of these problems, we use linear hash chaining combined with indirect hashing. When the load of the hash table reaches a certain threshold, we rehash into a larger table.

The user can increase the initial size of the hash table to save on resizing and rehashing operations. The ATerm library provides facilities for defining hash tables as well. This allows the implementation of a fast lookup mechanism for ATerms. User-defined hash tables are used, for instance, to implement memo-functions in the ASF+SDF to C compiler (see Section 6.5.3).

6.3.3 Garbage Collection

Which Technique? The most common strategies for automatic recycling of unused space are reference counting, mark-compact collection, and mark-sweep collection. In our case, reference counting is not a valid alternative, because it takes too much time and space and is very hard to implement in C. Mark compact garbage collection is also unattractive because it assumes that objects can be relocated. This is not the case in C where we cannot identify *all* references to an object. We can only determine the root set conservatively which is good enough for mark-sweep collection discussed below, but not for mark-compact collection.

Mark-sweep Garbage Collection Mark-sweep garbage collection is usually based on three phases. In the first phase, all objects on the heap are marked as 'dead'. In the second phase, all objects reachable from the known set of root objects are marked as 'live'. In the third phase, all 'dead' objects are swept into a list of free objects.

Mark-sweep garbage collection can be implemented in C efficiently, and without support from the programmer or compiler [BW88, Boe93a]. Mark-sweep collection is more efficient, both in time and space than reference counting [JL96]. A possible drawback is increased memory fragmentation compared to mark-compact collection. The typical space overhead for a mark-sweep garbage collection algorithm is only 1 bit per object, whereas a reference count field would take at least three or four bytes.

Reusing an Existing Garbage Collector A number of excellent generic garbage collectors for C are freely available, so why do we not reuse an existing implementation?

We have examined a number of alternatives, but none of them fit our needs. The Boehm-Weiser garbage collector [BW88] came close, but we face a number of unusual circumstances that render existing garbage collectors impractical:

- The hash table always contains references to all objects. It must be possible to instruct the garbage collector not to scan this area for roots.
- After an object becomes garbage, it must also be removed from the hash table. This means that we need very low level control over the garbage collector.
- The ATerm data type has some special characteristics that can be exploited to dramatically increase performance:
  - Destructive updates are not allowed. In garbage collection terminology, this means that there are no pointers from old objects to younger objects. Although we do not exploit it in the current implementation, this characteristic makes the use of a *generational* garbage collector very attractive.
  - The majority of objects have an in-memory representation of 8, 12, or 16 bytes.
  - Practical experience has shown that not many root pointers are kept in static variables or on the generic C heap. Performance can be increased dramatically if we eliminate the expensive scan through the heap and the static data area for root pointers. The only downside is that we require the programmer to explicitly supply the set of roots that is located on the heap or in static variables.

These observations allow us to gain efficiency on several levels, using everything from low level system 'hacks' to high-level optimizations.

Implementing the Garbage Collector Note that only ATerm objects are collected, Other objects, for instance objects allocated using `malloc` and friends, are not collected by our garbage collector. Considering both performance and the maintainability of the code that uses the ATerm library, we have opted for a version of the mark-sweep garbage collector. Every object contains a single bit used by the mark-sweep algorithm to indicate 'live' (marked) objects. At the start of a garbage collection cycle, all objects are unmarked. The garbage collector tries to locate and mark all live objects by traversing all terms that are explicitly protected by the programmer (using the `ATprotect` function), and by scanning the C run-time stack looking for words that could be references to objects. When such a word is found, the object (and the transitive closure of all of the objects it refers to) are marked as 'live'.

The scan of the run-time stack causes all objects referenced from local variables to be protected from being garbage collected. The programmer has the obligation to protect all other ATerm references explicitly: global, static, and heap variables. Although it would be possible to eliminate this obligation by also scanning the entire heap and static data areas for ATerm references, we believe that the performance gains outweigh the extra complexity introduced by the protect mechanism.

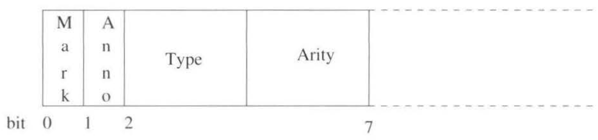


Figure 6.1: The header layout

Our garbage collector is a conservative collector in the sense that some of the words on the stack could accidentally have the same bit pattern as object references. Because there is no way to separate these 'fake' bit patterns from 'real' object references, this can cause objects to be marked as 'live' when these are actually garbage. Note that bit patterns on the stack that do not point to valid objects are not traversed at all. Only when a bit pattern represents an address that is a valid object address it is followed to mark the corresponding object.

When all live objects are marked, a single sweep through the heap is used to store all objects that are free in separate lists of free objects, one list for each object size.

As we shall see in Section 6.3.4, most objects consist of only a couple of machine words. By restricting the maximum arity of a function, we can also set an upper bound on the maximum size of objects. This enables us to base the memory management algorithms we use on a small number of block sizes. Allocation of objects is now simply a matter of taking the first element from the appropriate free-list, which is an extremely cheap operation. If garbage collection does not yield enough free objects, new memory blocks will be allocated to satisfy allocation requests.

6.3.4 Term Encoding

An important issue in the implementation of ATerms is how to represent this data type so that all operations can be performed efficiently in time and space.

The very concise encoding of ATerms we use is as follows. Assume that one machine word consists of four bytes. Every ATerm object is stored in two or more machine words. The first byte of the first word is called the *header* of the object, and consists of four fields (see Figure 6.1):

- A field consisting of one bit used as a mark flag by the garbage collector.
- A field consisting of one bit indicating whether or not this term has an annotation.
- A field consisting of three bits that indicate the type of the term.
- A field consisting of three bits representing the arity (number of pointers to other terms) of this object. When this field contains the maximum value of 7, the term must be a function application and the actual arity can be found by retrieving the arity of the function symbol (see below).

Depending on the type of the node (as determined by the header byte in the first word) the remaining bytes in the first word contain either a function symbol, a length indication, or they are unused.

The *second* word is always used for hashing, and links together all terms in the same hash bucket.

The type of the node determines its exact layout and contents. Figure 6.2 shows the encoding of the different term types which we will now describe in more detail.

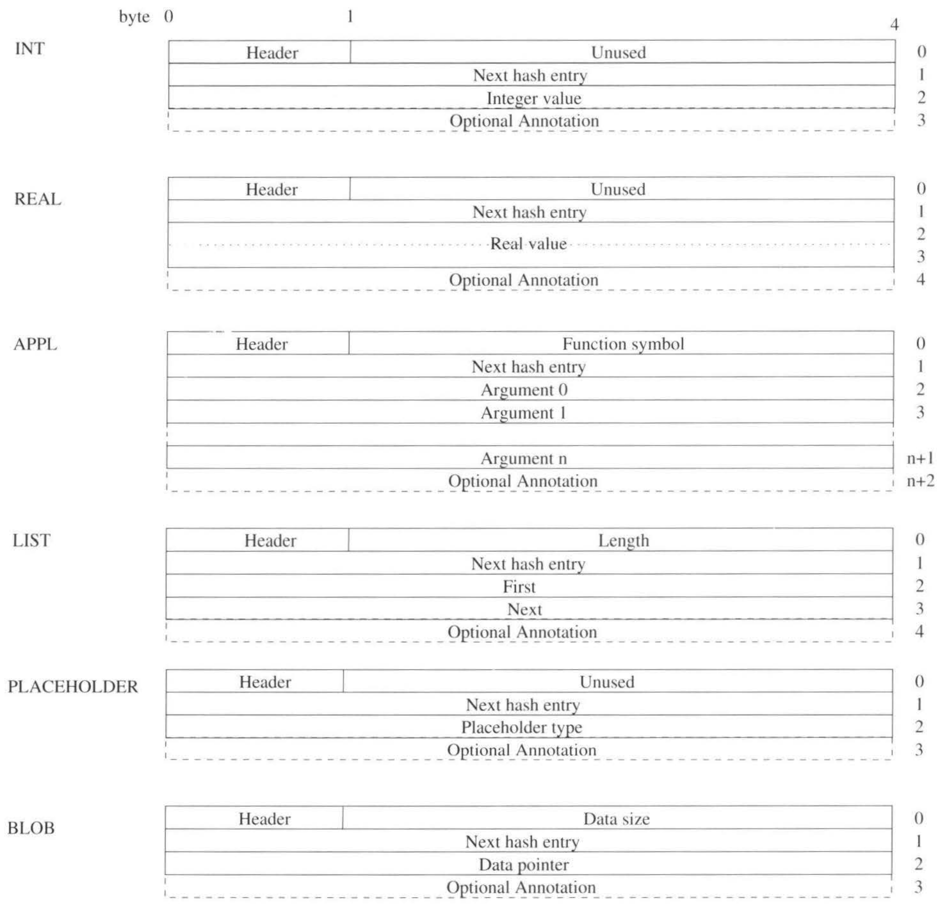


Figure 6.2: Encoding of the different term types

INT encoding In an integer term, the third word contains the integer value. The arity of an integer term is 0.

REAL encoding In an real term, the third and fourth word contain the real value represented by an 8 byte IEEE floating point number. The arity of a real term is 0.

APPL encoding The remaining 3 bytes following the header in the first word are used to represent the index in a table containing the function symbols. The words following the second word contain references to the function arguments. In this way, function applications can be encoded in $2 + n$ machine words, with n the arity of the function application.

LIST encoding The binary list constructor can be seen as a special function application with no function symbol and an arity of 2. The third word points to the first element in the list, this is called the `first` field, the fourth word points to the remainder of the list, and is called the `next` field. The length of the list is stored in the three bytes after the header in the first word. The empty list[5] is represented using a LIST object with empty first and next fields, and a length of 0.

After the function application, the list construct is the second most used ATerm construct. A (memory) efficient representation of lists is therefore very important. Due to the nature of the operations on ATerm lists, there are two obvious list representations: an array of term references or a linked list of term references. Experiments have shown that in typical applications quite varying list sizes are encountered. This renders the array approach inferior, because adding and deleting elements of a list would become too expensive. Consequently, we have opted for the linked list approach. Lists are constructed using binary list constructors, containing a reference to the first element in the list and to the tail of the list. Each list operation must ensure that the list is "normalized" again. This makes it very easy to perform the most commonly used operations on list, namely adding or removing the first element of a list.

Other operations are more expensive, since we do not allow destructive updates. Adding an element to the tail of a list for instance, requires n list creation operations, where n is the number of elements in the newly created list.

PLACEHOLDER encoding The placeholder term has an arity of 1, where the third word contains a pointer to the placeholder type.

BLOB encoding The length of the data contained in a BLOB term is stored in the three bytes after the header. This means that up to 16,777,200 bytes can be encoded in a single BLOB term. A pointer to the actual data is stored in the third word.

Annotations In all cases, annotations are represented using an extra word at the end of the term object. The single annotation bit in the header indicates whether or not an annotation is present. Only when this bit is set, an extra word is allocated that points to a term with type `LIST`, which represents the list of annotations.

[5]Due to the uniqueness of terms, only one instance of the empty list is present at any time.

6.3.5 ATerm Exchange: the Binary ATerm Format

The efficient exchange of ATerms between tools is very important. The simplest form of exchange is based on the concrete syntax presented in Appendix D. This would involve printing the term on one side and parsing it on the other. The concrete syntax is not a very efficient exchange format however, because the sharing of function symbols and subterms cannot be expressed in this way.

A better solution would be to exchange a representation in which sharing (both of function symbols and subterms) can be expressed concisely. A raw memory dump cannot be used, because addresses in the address space of one process have no meaning in the address space of another process.

In order to address these problems, we have developed BAF, the **B**inary **A**Term **F**ormat. Instead of writing addresses, we assign a unique number (index) to each subterm and each symbol occurring in a term that we want to exchange. When referring to this term, we could use its index instead of its address.

When writing a term, we begin by writing a table (in order of increasing indices) of all function symbols used in this term. Each function symbol consists of the string representation of its name followed by its arity.

ATerms are written in prefix order. To write a function application, first the index of the function symbol is written. Then the indices of the arguments are written. When an argument consists of a term that has not been written yet, the index of the argument is first written itself before continuing with the next argument. In this way, every subterm is written exactly once. Every time a parent term wishes to refer to a subterm, it just uses the subterm's index.

Exploiting ATerm Regularities When sending a large term containing many subterms, the subterm indices can become quite large. Consequently many bits are needed to represent these indices. We can considerably reduce the size of these indices when we take into account some of the regularities in the structure of terms. Empirical study shows that the set of function symbols that can actually occur at each of the argument positions of a function application with a given function symbol is often very small. An explanation for this is that although ATerm applications themselves are not typed, the data types they represent often are. In this case, function applications represent objects and the type of the object is represented by the function symbol. The type hierarchy determines which types can occur at each position in the object.

We exploit this knowledge by grouping all terms according to their top function symbol. Terms that are not function applications are grouped based on dummy function symbols, one for each term type. For each function symbol, we determine which function symbols can occur at each argument position. When writing the table of function symbols at the start of the BAF file, we write this information as well. In most cases this number of function symbol occurrences is very small compared to the number of terms that is to be written. Storing some extra information for every function symbol in order to get better compression is therefore worthwhile.

When writing the argument of a function application, we start by writing the actual symbol of the argument. Because this symbol is taken from a limited set of function symbols (only those symbols that can actually occur at this position), we can use a very small number to represent it. Following this function symbol we write the index of the argument term itself in the table of terms over this function symbol instead of the index of the argument in the total term table.

Example As an example, we show how the term `mult(s(s(z)),s(z))` is represented in BAF. This term contains three function symbols: `mult` with arity two, `s` with arity one, and `z` with arity zero. When grouping the subterms by function symbol we get:

0: `mult`	1: `s`	2: `z`
`mult(s(s(z)),s(z))`	`s(s(z))`	`z`
	`s(z)`	

When we look at the function symbols that can occur at every argument position (≥ 0) we get:

position	`mult`	`s`	`z`
0	`s`	`s, z`	
1	`s`		

We start by writing this symbol information to file. To do this, we have to write the following bytes[6]:

4 `"mult"`	: The length (`4`) and ASCII representation of `mult`.
2	: The arity (`2`) of `mult`.
1 1	: There is only one symbol (`1`) that can occur at the first argument position of `mult`. This is symbol `s` with index (`1`)
1 1	: At the second argument position, there is only (`1`) possible top symbol and that is `s` with index (`1`).
1 `"s"`	: The length (`1`) and ASCII representation of `s`.
1	: The arity (`1`) of `s`.
2 1 2	: The single argument of `s` can be either of two (`2`) different top function symbols: `s` with index (`1`) or `z` with index (`2`).

[6]When the value of these numbers used exceeds 127, two or more bytes are used to encode them. Strings are written as strings to improve readability.

1 "z"	: The length (1) and ASCII representation of z.
0	: The arity (0) of z.

Following this symbol information, the actual term `mult(s(s(z)),s(z))` can be encoded using only a handful of bits. Note that the first function symbol in the symbol table is always the top function symbol of the term (in this case: `mult`):

	: No bits need to be written to identify the function symbol `s`, because it is the only possible function symbol at the first argument position of `mult`.
0	: One bit indicates which term over the function symbol `s` is written (`s(s(z))`). Because this term has not been written yet, it is done so now.
0	: The function symbol of the only argument of `s(s(z))` is `s`.
1	: `s(z)` has index 1 in the term table of symbol `s`.
1	: Symbol `z` has index 1 in the symbol table of symbol `s`.
	: Because there is only one term over symbol `z`, no bits are needed to encode this term. Now we only need to encode the second argument of the input term, `s(z)`.
	: No bits are needed to encode the function symbol `s`, because it is the only symbol that can occur as the second argument of mult.
1	: `s(z)` has index 1 in the term table of symbol `s`. Because this term has already been written, we are done.

Only five bits are thus needed to encode the term `mult(s(s(z)),s(z))`. As mentioned earlier, the amount of data needed to write the table of function symbols at the start of the BAF file is in most cases negligible compared to the actual term data.

6.4 Performance Measurements

6.4.1 Benchmarks

How concise is the ATerm representation and how fast can BAF files be read and written? Since results highly depend on the actual terms being used, we will base our measurements on a collection of terms that cover most applications we have encountered so far.

Artificial Cases Two artificial cases are used that have been constructed to act as borderline cases:

Random-unique: a randomly generated term over a signature of 9 fixed function symbols with arities ranging from 1 to 9 and an arbitrary number of constant symbols (functions with arity 0). The terms are generated in such a way that all constants are unique. These terms are the worst case for our implementation: there is no regularity to exploit and there are many subterms with a relatively high arity.

Random: a randomly generated term over a signature of 10 function symbols with arities ranging from 0 to 9. In these terms only a single constant can occur which will be shared, but no other regularities can be exploited and there are many subterms with a relatively high arity.

Real Cases Several real-life cases are used that are based on actual applications:

COBOL Parse Table: a generated parse table for COBOL including embedded SQL and CICS. The grammar consists of 2,009 productions and the generated automaton has 6,699 states. The parse table contains an action-table (20,947 non-empty entries) and a goto-table (76,527 non-empty entries). This is an example of an abstract data type represented as ATerm.

COBOL System: a COBOL system consisting of 117 programs with a total of 247,548 lines of COBOL source code. It has been parsed with the above parse table. The parse trees constructed for these COBOL programs are represented as ATerms, see Section 6.5.1 for more details.

Risla Library: a parse tree of the component library for the Risla language, a domain specific language for describing financial products [ADR95]. This component library consists of 10,832 lines of code.

LPO: a linear process operator (LPO) describing the "firewire" protocol with 1 bus and 9 links [GL99, Lut99]. LPOs are the kernel of the μCRL ToolKit [DG95] which is a collection of tools for the manipulation of process and data descriptions in μCRL (micro Common Representation Language) [GP95]. An LPO is a structured process, where the state consists of an assignment to a sequence of typed data variables and its behavior is described by condition, action and effect functions. These states are represented as ATerms, and are rather complex.

Casl specifications: a collection of abstract syntax trees represented as ATerms of 98 Casl files, the total number of lines of Casl code is 2,506. For more details on Casl and the abstract syntax tree representation as ATerms we refer to Section 6.5.1.

lcc Parse Forest: a new back-end similar to the ASDL back-end [WAKS97] has been added to the lcc compiler [Han99a]. This back-end maps the internal format used by the lcc compiler to ATerms. The ATerm representation and the ASDL representation of a C program contain equivalent information.

Term	# nodes	# unique nodes	Sharing (%)	Memory (bytes)	Bytes/ Node
Artificial Cases					
Random-unique	1,000,000	1,000,000	0.00	15,198,694	15.20
Random	1,000,000	92,246	90.81	2,997,120	3.00
Real Cases					
COBOL Parse Table	961,070	97,516	89.85	2,836,529	2.95
COBOL System	31,332,871	470,872	98.50	12,896,609	0.41
Risla Library	708,838	40,073	94.35	960,170	1.35
LPO	8,894,391	225,229	97.47	3,701,438	0.42
Casl Specifications	34,526	11,699	66.12	235,655	6.83
lcc Parse Forest	360,829	86,589	76.00	1,547,713	4.29
S-expressions	593,874	283,891	52.20	9,111,863	15.34
Real Case Averages			82.07		4.51

Table 6.1: Memory usage of ATerms

Given this back-end the C sources of the lcc compiler itself are mapped to ATerms. The lcc compiler consists of 34 source files, consisting of a total of 13,588 lines of source code.

S-expressions: a simple translator has been developed which transforms an S-expression into an ATerm. This translator has been used to process an arbitrary collection of ".el" files containing S-expressions found within the Emacs source tree under Linux. The total number of ".el" files was 738, these files together contained 286,973 lines of code.

In the cases of the COBOL System, Casl Specifications, lcc Parse Forest, and S-Expressions the set of ATerms are combined into and processed as *one* ATerm. Measurements were performed on an ULTRA SPARC-5 (270 MHz) with 256 Mb of memory. All times measured are the user CPU time for that particular job.

6.4.2 Measurements

In Table 6.1, we give results for the memory usage of our sample terms[7]. The five columns give the total number of nodes in each term, the number of unique nodes in each term, the sharing percentage, the amount of memory (in bytes) used for the storage of the term, and the average number of bytes needed per node. As can be seen in these figures, at least in

[7]Since we consider the Random-unique and Random cases to be unrepresentative, we only present the averages for the real cases in this and the following tables.

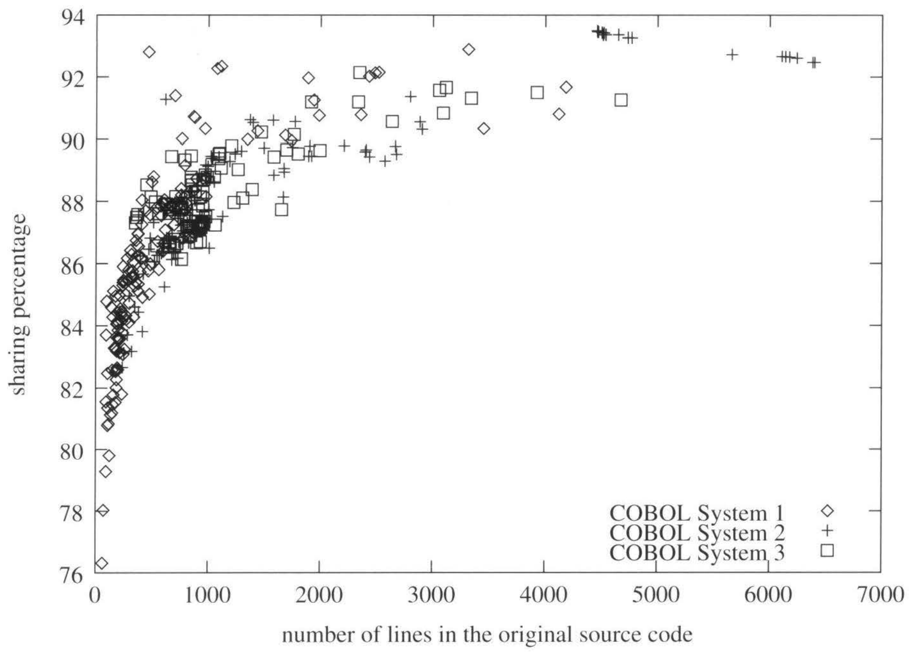


Figure 6.3: Sharing of a large number of COBOL parse trees

our applications sharing *does* make a difference. By fully exploiting the redundancies in the input terms, we can store a node using on the average 4.5 bytes, and still perform operations on them efficiently. The worst case behaviour is 15 bytes per node. The amount of sharing is clearly less high in case of abstract syntax trees than in case of parse trees represented as AsFix terms. The AsFix terms contain much redundant information which can be optimally shared. The amount of sharing in the abstract syntax trees for Casl is lower, but this is due to the fact that the set of Casl specifications is small and each specification tests another feature of the Casl language, so not much sharing was to be expected. The S-expressions have the lowest ratio of sharing, but this was to be expected: they represent ad hoc *hand-written* Lisp programs while in the other cases the ATerms are obtained by a systematic translation from source code. In the latter case, recurring patterns in the translation scheme result in higher levels of sharing.

Figure 6.3 shows the amount of sharing with respect to the size of a large number of COBOL programs. Three different sets of COBOL programs were considered. The first system consists of 151 files, the second of 116 files, and the last of 98 files. From this figure it can be concluded that the amount of sharing increases with the size of the COBOL system. In all three systems, the percentage of sharing converges to slightly over 90%. We find this

Term	ASCII (bytes)	Bytes/ Node	Read (s)	Read/ Node (μs)	Write (s)	Write/ Node (μs)
Artificial Cases						
Random-unique	6,888,889	6.89	34.76	34.76	4.06	4.06
Random	6,200,251	6.20	15.90	15.90	3.67	3.67
Real Cases						
COBOL Parse Table	4,211,366	4.38	6.33	6.95	2.30	2.29
COBOL System	135,350,005	4.32	199.43	6.36	65.02	2.08
Risla Library	2,955,964	4.17	4.25	6.00	1.40	1.98
LPO	41,227,481	4.64	81.90	9.21	29.16	3.28
Casl Specifications	217,958	6.31	0.36	10.43	0.08	2.32
lcc Parse Fores	2,132,245	6.22	3.13	9.14	0.86	2.51
S-expressions	7,954,550	13.39	15.09	25.41	2.49	4.19
Real Case Averages		6.20		10.50		2.66

Table 6.2: Reading and writing ATerms as ASCII text

high percentage in combination with the strong correlation between size and sharing very remarkable and will analyze its causes and consequences in further detail in a separate paper.

In Table 6.2 we give results for reading and writing our sample terms as ASCII text files. The six columns give the size of the text representation of the test term in bytes, the average number of bytes per node, the time needed to read the text file, the average time needed to read a node, the time needed to write the text file, and the average time needed to write a node. On the average, a node requires 6.2 bytes and reading and writing requires 10.5 μs and 2.7 μs, respectively.

In Table 6.3 we give results for reading and writing BAF files for the same set of sample terms. The columns give in order: the size of the BAF files in bytes, the average number of bytes needed per node, the time to read the BAF representation, the average read time per node, the time to write the BAF representation, and the average write time per node. Typically, we can read a node in 1.3 μs and write it in 2.4 μs.

Note that reading a BAF term is faster than writing the same term, whereas in case of ASCII the writing is faster than reading. This is caused by the fact that reading the ASCII representation of an ATerm involves numerous matching operations, whereas reading the BAF representation can be done with less matching. On the other hand, writing the BAF representation involves more calculations to encode the sharing of terms, whereas writing the ASCII representation involves a straightforward term traversal.

In Table 6.4 we show how the compression in BAF files compares to the compression of the standard Unix utility `gzip`. Considering the same set of examples, we give figures for

Term	BAF (bytes)	Bytes/ Node	Read (s)	Read/ Node (μs)	Write (s)	Write/ Node (μs)
Artificial Cases						
Random-unique	6,073,795	6.07	8.85	8.85	11.57	11.57
Random	567,419	0.57	2.06	2.06	2.76	2.76
Real Cases						
COBOL Parse Table	370,450	0.39	0.63	0.66	1.75	1.82
COBOL System	2,279,066	0.07	4.88	0.16	20.76	0.66
Risla Library	141,946	0.20	0.22	0.31	0.75	1.06
LPO	1,106,661	0.12	1.86	0.21	9.40	1.06
Casl Specifications	32,083	0.93	0.05	1.45	0.15	4.34
lcc Parse Forest	358,318	0.99	0.34	0.99	0.95	2.77
S-expressions	4,438,229	7.47	3.31	5.57	10.49	6.23
Real Case Averages		1.45		1.32		2.42

Table 6.3: Reading and writing ATerms as BAF

Term	ASCII (bytes)	BAF (bytes)	Comp. (%)	`gzip` (bytes)	Comp. (%)
Artificial Cases					
Random-unique	6,888,889	6,073,795	11.8	2,324,804	66.3
Random	6,199,981	567,419	90.9	439,293	92.9
Real Cases					
COBOL Parse Table	4,211,366	370,450	91.2	230,297	94.5
COBOL System	135,350,005	2,279,066	98.3	3,072,774	97.7
Risla Library	2,955,964	141,946	95.2	80,009	97.3
LPO	41,227,481	1,106,661	97.3	804,521	98.0
Casl Specifications	217,958	32,083	85.3	20,767	90.5
lcc Parse Forest	2,244,691	358,318	84.0	244,502	89.1
S-expressions	7,954,550	4,438,229	44.2	1,858,366	76.6
Real Case Averages			85.1		92.0

Table 6.4: BAF *versus* `gzip`

a straightforward dump of each term as ASCII text (column 1), the size of the BAF version of the same term (column 2) and percentage of compression achieved (column 3). Next, we give the results of compressing the ASCII version of each term with `gzip` (column 4), and compression achieved (column 5). The compression factors are 85% for BAF and 92% for `gzip`. The worst case compression of `gzip` (66%) is considerably better than the worst case compression using BAF (12%). No gains are to be expected from using `gzip` instead of BAF, since this would imply first writing the ATerm in textual format (an expensive operation which looses sharing) and then compressing it.

	Memory	ASCII	BAF
Size per node (bytes)	4.51	6.20	1.45
Read node (μs)		10.50	1.32
Write node (μs)		2.66	2.42

Table 6.5: Summary of measurements (based on Real Case averages)

6.4.3 Summary of Measurements

These measurements are summarized in Table 6.5. For in-memory storage, 4.5 bytes are needed per node. Using BAF, only 1.54 bytes are needed to represent a node. Also observe that reading BAF is an order of magnitude faster than reading terms in textual form. In case of parse trees represented as AsFix (COBOL System and Risla Library) less than 2 bytes are needed to represent a node in memory and less than 2 bits (0.20 bytes) are needed to represent it in binary format.

6.5 Applications

ATerms have already been used in applications ranging from development tools for domain specific languages [DK98] to factories for the renovation of COBOL programs [BSV97]. The ATerm data type is also the basic data type to represent the terms manipulated by the rewrite engines generated by the ASF+SDF compiler [BKO99] and they play a central role in the development of the new ASF+SDF Meta-Environment [BKMO97].

6.5.1 Representing Syntax Trees: AsFix and CasFix

The ATerm data type proves to be a powerful and flexible mechanism to represent syntax trees. By defining an appropriate set of function symbols parse trees and abstract syntax trees can be represented for any language or formalism. We describe two examples: AsFix (a parse tree format for ASF+SDF, Section 6.5.1) and CasFix (an abstract syntax tree format for Casl, Section 6.5.1).

AsFix The AsFix format (ASF+SDF Fixed format) is an incarnation of ATerms for representing ASF+SDF [HHKR92, BHK89, DHK96]. ASF+SDF is a modular algebraic specification formalism for describing the syntax and semantics of (programming) languages. SDF (Syntax Definition Formalism) allows the definition of the concrete and abstract syntax of a language and is comparable to (E)BNF. ASF (Algebraic Specification Formalism) allows the definition of the semantics in terms of equations, which are interpreted as rewrite rules. The development of ASF+SDF specifications is supported by an integrated programming environment, the ASF+SDF Meta-Environment [Kli93].

Using AsFix, each module or term is represented by its parse tree which contains both the syntax rules used and all original layout and comments. In this way, the original source text can be reconstructed from the AsFix representation, thus enabling transformation tools to access and transform comments in the source text. Since the AsFix representation is self-contained (all grammar information needed to interpret the term is also included), one can easily develop tools for processing AsFix terms which do not have to consult a common database with grammar information. Examples of such tools are a (structure) editor or a rewrite engine.

AsFix is defined by an appropriate set of function symbols for representing common constructs in a parse tree. These function symbols include the following:

- `prod`(T) represents production rule T.
- `appl`(T_1, T_2) represents applying production rule T_1 to the arguments T_2.
- `l`(T) represents literal T.
- `sort`(T) represents sort T.
- `lex`(T_1, T_2) represents (lexical) token T_1 of sort T_2.
- `w`(T) represents white space T.
- `attr`(T) represents a single attribute.
- `attrs`(T) represents a list of attributes.
- `no-attrs` represents an empty list of attributes.

The following context-free syntax rules (in SDF [HHKR92]) are necessary to parse the input sentence `true or false`.

```
sort Bool
context-free syntax
  true         -> Bool
  false        -> Bool
  Bool or Bool -> Bool {left}
```

The parse tree below gives the AsFix representation for the input sentence `true or false`.

```
appl(prod([sort("Bool"),l("or"),sort("Bool")],sort("Bool"),
          attrs([attr("left")])),
      [appl(prod([l("true")],sort("Bool"),no-attrs),[l("true")]),
       w(" "),l("or"),w(" "),
       appl(prod([l("false")],sort("Bool"),no-attrs),[l("false")])
      ])
```

Two observations can be made about this parse tree. First, this parse tree is an ordinary ATerm, and can be manipulated by all ATerm utilities in a completely generic way.

Second, this parse tree is completely self-contained and does not depend on a separate grammar definition. It is clear that this way of representing parse trees contains much redundant information. Therefore, both maximal sharing and BAF are essential to reduce their size. In our measurements, AsFix only plays a role in the cases COBOL System and Risla Library.

The annotations provided by the ATerm data type can be used to store auxiliary information like positional information derived by the parser or font and/or color information needed by a (structure) editor. This information is globally available but can be ignored by tools that are not interested in it.

CasFix Casl (Common Algebraic Specification Language) is a new algebraic specification formalism [CL98] developed as part of the CoFI initiative. It is a general algebraic specification formalism incorporating common features of most existing algebraic specification languages. In addition to the language itself, a set of tools is planned for supporting the development of Casl specifications. Existing tools will be reused as much as possible.

In order to let the various tools, like parsers, editors, rewriters, and proof checkers, communicate with each other an intermediate format was needed for Casl. ATerms have been selected as intermediate format and a specialized version for representing the *abstract* syntax trees of Casl has been designed (CasFix [BKO98]). Contrast this with the approach taken for AsFix, where the more concrete *parse trees* are used as intermediate representation.

CasFix is obtained by defining an appropriate set of function symbols for representing Casl's abstract syntax [CL98] and by defining a mapping from Casl's concrete syntax to its abstract syntax. For each abstract syntax rule an equivalent CasFix construct is defined as in:

```
ALTERNATIVE ::= "total-construct" OP-NAME COMPONENTS*
```

$\Longrightarrow$

```
total-construct(<OP-NAME>,COMPONENTS*([<COMPONENTS>]))
```

In this example `"total-construct"` and `"COMPONENTS*"` are function symbols and `<OP-NAME>` and `<COMPONENTS>` represent the subtrees of the corresponding sort.

6.5.2 ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [Kli93] is an interactive development environment for writing language specifications in ASF+SDF. A new generation of this environment is being developed based on separate components connected via the TOOLBUS [BK98]. A description of this new architecture can be found in [BKMO97]. The new ASF+SDF Meta-Environment provides tools for parsing, compilation, rewriting, debugging, and formatting. ATerms and AsFix play an important role in the new ASF+SDF Meta-Environment:

- The parser generator [Vis97] produces a parse table represented as ATerm.
- The parser uses this parse table and transforms an input string into a parse tree which is represented as AsFix term.
- After parsing, the modules of an ASF+SDF specification are stored as AsFix terms. Information concerning the specification such as the rewrite rules that must be compiled are exchanged as AsFix terms.
- The ASF+SDF compiler (see next section) reads and writes AsFix terms.

6.5.3 ASF+SDF to C compiler

The ASF+SDF to C compiler [BKO99] is a compiler for ASF+SDF. It generates ANSI-C code and depends on the ATerm library as run-time environment. All terms manipulated by the generated C code are represented as ATerms thus taking advantage of maximal subterm sharing and automatic garbage collection.

The optimized memory usage of ATerms has already been exploited in various industrial projects [BDK$^+$96, BKV98] where memory usage is a critical success factor. This ASF+SDF compiler has, for instance, been applied successfully in projects such as the development of a domain-specific language for describing interest products (in the financial domain) [ADR95] and a renovation factory for restructuring COBOL code [BSV97].

The ASF+SDF compiler is an ASF+SDF specification and has been bootstrapped. Table 6.6 gives some figures on the size of this specification and the time needed to compile it. Table 6.7 gives an impression of the effect of compiling the ASF+SDF compiler with and without sharing. More information on the compiler itself and on performance issues can be found in [BKO99].

6.5.4 Other Applications

Other applications are still under development and include:

- A tool for protocol verification [GL99]. The ATerms are used to represent the states in the state space of the protocol. Because of the huge amount of states ($\geq 1,000,000$) it is necessary to share as many states as possible.

Specification	ASF+SDF (equations)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF compiler (sec)	C compiler (sec)
ASF+SDF compiler	1,876	8,699	85,185	216	323

Table 6.6: Some figures on the ASF+SDF compiler

Application	Time (sec)	Memory (Mb)
ASF+SDF compiler (with sharing)	216	16
ASF+SDF compiler (without sharing)	661	117

Table 6.7: Performance with and without maximal sharing

- A tool for the detection of code clones in legacy code.
- The Stratego compiler [VBT98].

6.6 Discussion

6.6.1 Related Work

S-expressions in LISP Many intermediate representations are derived in some form or another from the S-expressions in LISP. ATerms are no exception to this rule. The main improvements of ATerms over S-expressions are

- ATerms support arbitrary binary data (Blobs, see Section 6.2.1).
- ATerms support annotations.
- ATerms support maximal sharing in a systematic way.
- ATerms support a concise, sharing preserving, exchange format that exploits the implicit signature of terms.
- The ATerm library provides a comprehensive collection of access functions based on the *match-and-make* paradigm.

Intermediate representations in compiler frameworks There exist numerous frameworks for compilers and programming environments that provide facilities for representing intermediate data. Examples are Centaur's VTP [BCD+89], Eli [GHL+92], Cocktail's Ast [Gro92], SUIF [WFW$^+$94], ASDL [WAKS97], and Montana [Kar98]. These systems either provide an explicit intermediate format (Eli, Ast, SUIF) or they provide a programmable interface to the intermediate data (VTP, Montana, ASDL). Lamb's IDL [Lam87] and OMG's IDL [OMG97] are frameworks for representing intermediate data that are not tied to a specific compiler construction paradigm but have objectives similar to the systems already mentioned.

These approaches typically use a grammar-like definition of the abstract syntax (including attributes) and provide (generated) access functions as well as readers and writers for these intermediate data. In most cases support exists for accessing the intermediate data from a small collection of source languages.

The major difference between these approaches and ATerms is that they operate at different levels of abstraction. ATerms just provide the lower-level representation for terms (or more precisely directed acyclic graphs), while intermediate representations for compilers are more specialized and give a higher-level view on the intermediate data. They provide primitives for representing program constructs, symbol tables, flow graphs and other derived information. In most cases they also provide a fixed format for representing programs at different levels of abstraction ranging from call graphs to machine-like instructions. ATerms are thus simpler and more general and they can be used to represent each of these compiler's intermediate formats.

Another difference is that most compiler frameworks use a statically typed intermediate representation. The major advantage is early error-detection. The disadvantages are, however, less flexibility and the need to generate different access functions for each different intermediate format. In the case of ATerms, a dynamic check may be necessary on the intermediate data but only a single, generic, set of access functions is needed.

ASDL The abstract syntax definition language (ASDL) [WAKS97] is a language for describing tree data structures and is used as intermediate representation language between the various phases of a compiler [Han99a]. We consider ASDL in more detail, because of its public availability and the fact that the goals of ASDL and ATerms are quite similar as they are both used to exchange of syntax trees between tools, although ATerms are more general in the sense that other types of information, such as unstructured binary objects and annotations, can also be represented as an ATerm. Everything that can be represented by a grammar can be represented in ATerms as well as ASDL.

ASDL pickles and the BAF format for ATerms are comparable with respect to functionality, both are binary representations of (among others) syntax trees. The pickle and unpickle functions are generated from the ASDL description and are thus application specific (this may be more efficient) whereas the reading and writing of BAF is entirely generic (this avoids the proliferation of versions).

ASDL and ATerms can be compared at two different levels:

- *Low level:* ASDL pickle versus plain ATerms. By providing an ASDL definition of ATerms we can compare the size of the same object as ATerm (ASCII and BAF) and as ASDL pickle. This is done in Table 6.8 for the COBOL Parse Table. In this case, the

Term	ASCII	BAF	ASDL pickle
COBOL Parse Table	4,211,366	370,450	5,262,426

Table 6.8: Sizes of the COBOL parse table (in bytes)

Term	ASCII	BAF	ASDL pickle
lcc Parse Forest	2,246,436	624,091	1,290,595

Table 6.9: Sizes of abstract syntax trees (in bytes)

representation in BAF is an order of magnitude smaller than the ASDL pickle. Note that ASDL was not designed as an extremely compact representation format. It is for instance possible to combine ASDL pickles by simply concatenating them, while this is not possible with BAF terms. In ASDL, an identifier is represented by a string *for each occurence* of the identifier. In contrast, BAF uses an identifier table and only uses indices in this table.

- *High level*: compare at the level of parse trees or abstract syntax trees. ASDL is typically used to represent abstract syntax trees while ATerms can be used to represent both as we have discussed in Section 6.5.1. To make a meaningful comparison, we compare the abstract syntax trees generated by the lcc back-end in ATerm format (both in ASCII and BAF) and the corresponding ASDL pickles. These figures are presented in Table 6.9 for the abstract syntax trees generated for the lcc source files. In this case the BAF representation is 2 times smaller than the ASDL pickle. Note that the figure for the BAF representation differs from the figure in Table 6.3, this is caused by the fact that in Table 6.3 all files are combined into one BAF term whereas in Table 6.9 each file is a separate BAF term and their sizes are added.

XML The Extensible Markup Language [XML98] is a recently standardized format for Web documents. Unlike HTML, XML makes a strict distinction between *content* and *presentation*. XML can be *extended* by adding user-defined *tags* to parts of a document and by defining the overall structure of the document thus enabling well-formedness checks on documents. Although the original objectives are completely different, there are striking similarities between ATerms and XML: both serve the representation of hierarchically structured data and both allow arbitrary extensions (adding tags *versus* adding function symbols). There is also a straightforward translation possible between ATerms and XML.

The main difference between the two is that XML is more verbose and does not provide a simple mechanism to represent sharing, whereas ATerms provide the BAF format. This may not be a problem for Web documents like catalogs and database records, but is does present a major obstacle in our case when we need to exchange huge terms between tools. We are currently considering whether some link between ATerms and XML may be advantageous.

Data encodings As described in Section 6.3.5, we use a form of data encoding to compress ATerms when they are exchanged between tools. Of course, encoding and data compression techniques are in common use in telecommunications. For instance, the ASN.1 standard gives detailed rules for data encoding [ASN95].

In an earlier project in our group, the Graph Exchange Language (GEL) [Kam94] has been developed. It is similar in goals to BAF, but BAF can only represent acyclic directed graphs, whereas GEL can represent arbitrary (potentially cyclic) graphs. The technical approaches are different as well. GEL uses a binary-encoded postfix format to represent the nodes in the graph and introduces explicit labels to reuse previously constructed parts of the graph. BAF uses a prefix format augmented by generated symbol tables.

A final difference is in the *usage* of both approaches. GEL was used as a separate library that could be used in applications and the graph encoding was therefore visible to the programmer using it. BAF is, on the other hand, completely integrated in the ATerm implementation and is only used by the standard read and write functions for ATerms. The BAF format is therefore never visible to programmers.

Hash consing In LISP, the success of hash consing [All78] has been limited by the existence of the functions `rplaca` and `rplacd` that can destructively modify a list structure. To support destructive updates, one has to support two kinds of list structures "mono copy" lists with maximal sharing and "multi copy" lists without maximal sharing. Before destructively changing a mono copy list, it has to be converted to a multi copy list. In the 1970's, E. Goto has experimented with a Lisp dialect (HLisp) supporting hash consing and list types as just sketched. See [TK90] for a recent overview of this work and its applications.

A striking observation can be made in the context of SML [AG93] where sharing resulted in slightly increased execution speed and only marginal space savings. On closer inspection, we come to the conclusion that both methods for term sharing are different and can not be compared easily. We share terms immediately when they are created: the costs are a table lookup and the storage needed for the table while the benefits are space savings due to sharing and a fast equality test (one pointer comparison). In [AG93] sharing of subterms is *only* determined during garbage collection in order to minimize the overhead of a table lookup at term creation. This implies that local terms that have not yet survived one garbage collection are not yet shared thus loosing most of the benefits (space savings and fast equality test) as well.

6.6.2 History

Terms are so simple that most programmers prefer to write their own implementation rather than using (or even *looking for*) an existing implementation. This is all right, except when this happens in a group of cooperating developers as in our case.

A very first version of the ATerm library was developed as part of the TOOLBUS coordination architecture [BK98]. It was used to represent data which were transported between tools written in different languages running on different machines. Simultaneously, we were developing a formalism for representing parse trees [GB94]. In addition, incompatible term formats were in use in various of our compiler projects [FKW98]. Observing the similarities between all these incompatible term data types triggered the work on ATerms as described here. The benefits are twofold. First, a common term data type is used in more applications and investments in it are well rewarded. Second, the mere existence of a common data type leads to new, unanticipated, applications. For instance, we now use ATerms for representing parse tables as well.

6.6.3 Conclusions

As stated in the introduction, ATerms are intended to form an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of (tree-like) data structures between distributed applications.

ATerms *are* open and language independent since they do not depend on any specific hardware or software platform. ATerms *are* simple: the level one interface consists of only 13 functions. ATerms *are* efficient and concise as shown by the measurements in Section 6.4. Last but not least, ATerms are also *useful* as shown on Section 6.5.

The ATerm format is supported by a binary exchange format (BAF) which provides a mechanism to exchange ATerms in a concise way. This BAF format maintains the in-memory sharing of terms and uses a minimal amount of bits to represent the nodes, in case of AsFix terms only 2 bits are needed per node.

The most innovative aspects of ATerms are the simple procedural interface based on the *make-and-match* paradigm, term annotations, maximal subterm sharing, and the concise binary encoding of terms that is completely hidden behind high-level read and write operations.

7 Compiling ASF+SDF specifications

We have developed a compiler for the ASF+SDF formalism that has been used successfully for several industrial applications.. This result is achieved in two ways: the compiler performs a variety of optimizations and generates efficient C code, and the compiled code uses a run-time memory management system based on the ATerm library discussed in the previous chapter. Without techniques like maximal subterm sharing and mark-and-sweep garbage collection we cannot achieve the performance required to reach our goals.

We present an overview of these techniques and evaluate their effectiveness in several benchmarks. It turns out that execution speed of compiled ASF+SDF specifications is at least as good as that of comparable systems, while memory usage is in many cases an order of magnitude smaller.

7.1 Introduction

Efficient implementation based on mainstream technology is a prerequisite for the application and acceptance of declarative languages or specification formalisms in real industrial settings. The main characteristic of industrial applications is their *size* and the predominant implementation consideration should therefore be the ability to handle huge problems.

In this chapter we take the specification formalism ASF+SDF as point of departure. Its main focus is on language prototyping and on the development of language specific tools. ASF+SDF is based on general context-free grammars for describing syntax and on conditional equations for describing semantics. In this way, one can easily describe the syntax of a (new or existing) language and specify operations on programs in that language such as static type checking, interpretation, compilation or transformation. ASF+SDF has been applied successfully in a number of industrial projects [BDK$^+$96, BKV98], such as the development of a domain-specific language for describing interest products (in the financial domain) [ADR95] and a renovation factory for restructuring of COBOL code [BSV97]. In such industrial applications, the execution speed is very important, but when processing huge COBOL programs memory usage becomes a critical issue as well. Other applications of ASF+SDF include the development of a GLR parser generator [Vis97], an unparser generator [BV96], program transformation tools [Bru96], and the compiler discussed in this chapter. Other components, such as parsers, structure editors, and interpreters, are developed in ASF+SDF as well but are not (yet) compiled to C.

What are the performance standards one should strive for when writing a compiler for, in our case, an algebraic specification formalism? Experimental, comparative, studies are scarce, one notable exception is [H$^+$96] where measurements are collected for various declarative programs solving a single real-world problem. In other studies it is no exception that the units of measurement (rewrite steps/second, or logical inferences/second) are ill-defined and that memory requirements are not considered due to the small size of the input problems.

In this chapter, we present a compiler for ASF+SDF that performs a variety of optimizations and generates efficient C code. The compiled code uses a run-time memory management system based on maximal subterm sharing and mark-and-sweep garbage collection. The contribution of this chapter is to bring the performance of executable specifications based on term rewriting into the realm of industrial applications.

In the following two subsections we will now first give a quick introduction to ASF+SDF (the input language of the compiler to be described) and to μASF (the abstract intermediate representation used internally by the compiler). Next, we describe the generation of C code (Section 7.2) as well as memory management (Section 7.3). Section 7.4 is devoted to benchmarking. A discussion in Section 7.5 concludes this chapter.

7.1.1 Specification Language: ASF+SDF

The specification formalism ASF+SDF [BHK89, HHKR92] is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. An overview can be found in [DHK96]. As an illustration, Figure 7.1 presents the definition of the Boolean datatype in ASF+SDF. ASF+SDF specifications consist of modules, each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations). The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation since arbitrary context-free grammars can be defined. The syntax defined in the SDF-part of a module can be used immediately when defining equations, the syntax in equations is thus *user-defined*.

The emphasis in this chapter will be on the compilation of the equations appearing in a specification. They have the following distinctive features:

- Conditional equations with positive and negative conditions.
- Non left-linear equations.
- List matching.
- Default equations.

```
imports Layout
exports
  sorts BOOL
  context-free syntax
    true                  → BOOL {constructor}
    false                 → BOOL {constructor}
    BOOL "|" BOOL         → BOOL {left}
    BOOL "&" BOOL         → BOOL {left}
    BOOL "xor" BOOL       → BOOL {left}
    not BOOL              → BOOL
    "(" BOOL ")"          → BOOL {bracket}
  variables
    Bool [0-9']* → BOOL
  priorities
    BOOL "|"BOOL → BOOL  <  BOOL "xor"BOOL → BOOL  <
    BOOL "&"BOOL → BOOL  <  not BOOL → BOOL
equations


[B1]  true | Bool      = true
[B2]  false | Bool     = Bool
[B3]  true & Bool      = Bool
[B4]  false & Bool     = false
[B5]  not false        = true
[B6]  not true         = false
[B7]  true xor Bool = not Bool
[B8]  false xor Bool = Bool
```

Figure 7.1: ASF+SDF specification of the Booleans

imports Layout
exports
  sorts ID
  lexical syntax
    [a-z][a-z0-9]* $\rightarrow$ ID
  sorts Set
  context-free syntax
    "{" {ID ","}* "}" $\rightarrow$ Set
hiddens
  variables
    Id "*"*[0-9]** $\rightarrow$ {ID ","}*
    *Id [0-9']** $\rightarrow$ ID
equations

[1] $\{Id_0^*, Id, Id_1^*, Id, Id_2^*\} = \{Id_0^*, Id, Id_1^*, Id_2^*\}$

Figure 7.2: ASF+SDF specification of the Set equation

It is possible to execute specifications by interpreting the equations as conditional rewrite rules. The semantics of ASF+SDF are based on innermost rewriting. Default equations are tried when all other applicable equations have failed, because either the arguments did not match or one of the conditions failed.

One of the powerful features of the ASF+SDF specification language is list matching. Figure 7.2 shows a single equation which removes multiple occurrences of identifiers from a set. In this example, variables with a *-superscript are list-variables that may match zero or more identifiers. The implementation of list matching may involve backtracking to find a match that satisfies the left-hand side of the rewrite rule as well as all its conditions. There is only backtracking within the scope of a rewrite rule, so if the right-hand side of the rewrite rule is normalized and this normalization fails *no* backtracking is performed to find a new match.

The development of ASF+SDF specifications is supported by an interactive programming environment, the ASF+SDF Meta-Environment [Kli93]. In this environment specifications can be developed and tested. It provides syntax-directed editors, a parser generator, and a rewrite engine. Given this rewrite engine terms can be reduced by interpreting the equations as rewrite rules. For instance, the term

```
true & ( false | true )
```

reduces to `true` when applying the equations of Figure 7.1.

7.1.2 Intermediate Representation Language: μASF

The user-defined syntax that may be used in equations poses two major implementation challenges.

First, how do we represent ASF+SDF specifications as parse trees? Recall that there is no fixed grammar since the basic ASF+SDF-grammar can be extended by the user. The solution we have adopted is to introduce the intermediate format AsFix (ASF+SDF fixed format) which is used to represent the parse trees of the ASF+SDF modules in a format that is simple to process by a machine. The user-defined syntax is replaced by prefix functions. The parse trees in the AsFix format are self contained.

Second, how do we represent ASF+SDF specifications in a more abstract form that is suitable as compiler input? We use a simplified language μASF as an intermediate representation to ease the compilation process and to perform various transformations before generating C code. μASF is in fact a single sorted (algebraic) specification formalism that uses only prefix notation. μASF can be considered as the abstract syntax representation of ASF+SDF. AsFix and μASF live on different levels, μASF is only visible within the compiler whereas AsFix serves as exchange format between the various components, such as structure editor, parser, and compiler.

A module in μASF consists of a module name, a list of functions, and a set of equations. The main differences between μASF and ASF+SDF are:

- Only prefix functions are used.
- The syntax is fixed (eliminating lexical and context-free definitions, priorities, and the like).
- Lists are represented by binary list constructors instead of the built-in list construct as in ASF+SDF; associative matching is used to implement list matching.
- Functions are untyped, only their arity is declared.
- Identifiers starting with capitals are variables; variable declarations are not needed.

Figure 7.3 shows the μASF specification corresponding to the ASF+SDF specification of the Booleans given earlier in Figure 7.1[1]. Figure 7.4 shows the μASF specification of sets given earlier in Figure 7.2. Note that this specification is not left-linear since the variable `Id` appears twice on the left-hand side of the equation. The `{list}` function is used to mark that a term is a list. This extra function is needed to distinguish between a single element list and an ordinary term, e.g., `{list}(a)` *versus* `a` or `{list}(V)` *versus* `V`. An example of a transformation on μASF specifications is shown in Figure 7.5, where the non-left-linearity has been removed from the specification in Figure 7.4 by introducing new variables and an auxiliary condition.

[1]To increase the readability of the generated code in this chapter, we have consistently renamed generated names by more readable ones, like `true`, `false`, etc.

```
module Booleans
signature
  true;
  false;
  and(_,_);
  or(_,_);
  xor(_,_);
  not(_);
rules
  and(true,B) = B;
  and(false,B) = false;
  or(true,B) = true;
  or(false,B) = B;
  not(true) = false;
  not(false) = true;
  xor(true,B) = not(B);
  xor(false,B) = B;
```

Figure 7.3: μASF specification of the Booleans

```
module Set
signature
  {list}(_);
  set(_);
  conc(_,_);
rules
  set({list}(conc(*Id0,conc(Id,conc(*Id1,conc(Id,*Id2)))))) =
    set({list}(conc(*Id0,conc(Id,conc(*Id1,*Id2)))));
```

Figure 7.4: μASF specification of Set

```
module Set
signature
  {list}(_);
  set(_);
  conc(_,_);
  t;
  term-equal(_,_);
rules
  term-equal(Id1,Id2) == t
  ==>
  set({list}(conc(*Id0,conc(Id1,conc(*Id1,conc(Id2,*Id2)))))) =
    set({list}(conc(*Id0,conc(Id1,conc(*Id1,*Id2)))));
```

Figure 7.5: Left-linear μASF specification of Set

7.2 C Code Generation

The ASF compiler uses μASF as intermediate representation format and generates C code as output. The compiler consists of several independent phases that gradually simplify and transform the μASF specification and finally generate C code.

A number of transformations is performed to eliminate "complex" features such as removal of non left-linear rewrite rules, simplification of matching patterns, and the introduction of "assignment" conditions (conditions that introduce new variable bindings). Some of these transformations are performed to improve the efficiency of the resulting code whereas others are performed to simplify code generation.

In the last phase of the compilation process C code is generated which implements the rewrite rules in the specification using adaptations of known techniques [Kap87, Dik89]. Care is taken in constructing an efficient matching automaton, identifying common and reusable (sub)expressions, and efficiently implementing list matching. For each μASF function (even the constructors) a separate C function is generated. The right-hand side of an equation is directly translated to a function call, if necessary. A detailed description of the construction of the matching automaton is beyond the scope of this chapter, a full description of the construction of the matching automaton can be found in [BHKO00]. Each generated C function contains a small part of the matching automaton, so instead of building one big automaton, the automaton is split over the functions. The matching automaton respects the syntactic specificity of the arguments from left to right in the left-hand sides of the equations. Non-variable arguments are tried before the variable ones.

The datatype `ATerm` (for Annotated Term) is the most important datatype used in the generated C code. It is provided by a run-time library which takes care of the creation, manipulation, and storage of terms. ATerms consist of a function symbol and zero or more

```
ATerm and(ATerm arg0, ATerm arg1) {
   if (check_sym(arg0, truesym))
     return arg1;
   if (check_sym(arg0, falsesym))
     return arg0;
   return make_nf2(andsym,arg0,arg1);
}
```

Figure 7.6: Generated C code for the and function of the Booleans

arguments, e.g., and(true, false). The library provides predicates, such as check_sym to check whether the function symbol of a term corresponds to the given function symbol, and functions, like make_nf$i$ to construct a term (normal form) given a function symbol and i arguments ($i \geq 0$). There are also access functions to obtain the i-th argument ($i \geq 0$) of a term, e.g., arg_1(and(true, false)) yields false.

The usage of these term manipulation functions can be seen in Figures 7.6 and 7.7. Figure 7.6 shows the C code generated for the and function of the Booleans (also see Figures 7.1 and 7.3). This C code also illustrates the detection of reusable subexpressions. In the second if-statement a check is made whether the first argument of the and-function is equal to the term false. If the outcome of this test is positive, the first argument arg0 of the and-function is returned rather than building a new normal form for the term false or calling the function false(). The last statement in Figure 7.6 is necessary to catch the case that the first argument is neither a true or false symbol, but some other Boolean normal form.

Figure 7.7 shows the C code generated for the Set example of Figure 7.2. List matching is translated into nested while loops, this is possible because of the restricted nature of the backtracking in list matching. The functions not_empty_list, list_head, list_tail, conc, and slice are library functions which give access to the C data structure which represents the ASF+SDF lists. In this way the generated C code needs no knowledge of the internal list structure. We can even change the internal representation of lists *without adapting the generated C code*, by just replacing the library functions. The function term_equal checks the equality of two terms.

When specifications grow larger, *separate compilation* becomes mandatory. There are two issues related to the separate compilation of ASF+SDF specifications that deserve special attention. The first issue concerns the identification and linking of names appearing in separately compiled modules. Essentially, this amounts to the question how to translate the ASF+SDF names into C names. This problem arises since a direct translation would generate names that are too long for C compilers and linkage editors. We have opted for a solution in which each generated C file contains a "register" function which stores at run-time for each defined function defined in this C file a mapping between the address of the generated function and the original ASF+SDF name. In addition, each C file contains a "resolve" function

```
ATerm set(ATerm arg0) {
   if(check_sym(arg0,listsym)) {
     ATerm tmp0 = arg_0(arg0);
     ATerm tmp1[2];
     tmp1[0] = tmp0;
     tmp1[1] = tmp0;
     while(not_empty_list(tmp0)) {
       ATerm tmp3 = list_head(tmp0);
       tmp0 = list_tail(tmp0);
       ATerm tmp2[2];
       tmp2[0] = tmp0;
       tmp2[1] = tmp0;
       while(not_empty_list(tmp0)) {
         ATerm tmp4 = list_head(tmp0);
         tmp0= list_tail(tmp0);
         if(term_equal(tmp3,tmp4)) {
           return set(list(conc(slice(tmp1[0],tmp1[1]),
                    conc(tmp3,conc(slice(tmp2[0],
                                tmp2[1]),tmp0)))));
         }
         tmp2[1] = list_tail(tmp2[1]);
         tmp0= tmp2[1];
       }
       tmp1[1] = list_tail(tmp1[1]);
       tmp0= tmp1[1];
     }
   }
   return make_nf1(setsym,arg0);
}
```

Figure 7.7: C code for the Set specification

which connects local function calls to the corresponding definitions based on their ASF+SDF names. An example of registering and resolving can be found in Figure 7.8.

The second issue concerns the choice of a unit for separate compilation. In most programming language environments, the basic compilation unit is a file. For example, a C source file can be compiled into an object file and several object files can be joined by the linkage editor into a single executable. If we change a statement in one of the source files, that complete source file has to be recompiled and linked with the other object files.

In the case of ASF+SDF, the natural compilation unit would be the module. However, we want to generate a single C function for each function in the specification (for efficiency reasons) but ASF+SDF functions can be defined in specifications using multiple equations

```
void register_xor() {
   xorsym = "prod(Bool xor Bool -> Bool {left})";
   register_prod("prod(Bool xor Bool -> Bool {left})",
                 xor, xorsym);
}

void resolve_xor() {
   true = lookup_func("prod(true -> Bool)");
   truesym = lookup_sym("prod(true -> Bool)");
   false = lookup_func("prod(false -> Bool)");
   falsesym = lookup_sym("prod(false -> Bool)");
   not = lookup_func("prod(not Bool -> Bool)");
   notsym = lookup_sym("prod(not Bool -> Bool)");
}

ATerm xor(ATerm arg0, ATerm arg1) {
   if (check_sym(arg0, truesym))
     return (*not)(arg1);
   if (check_sym(arg0, falsesym))
     return arg1;
   return make_nf2(xorsym,arg0,arg1);
}
```

Figure 7.8: Generated C code for the `xor` function of the Booleans

occurring in several modules. The solution is to use a single function as compilation unit and to *re-shuffle* the equations before translating the specification. Equations are thus stored depending on the module they occur in as well as on their outermost function symbol. When the user changes an equation, only those functions that are actually affected have to be re-compiled into C code. The resulting C code is then compiled, and linked together with all other previously compiled functions.

7.3 Memory Management

At run-time, the main activities of compiled ASF+SDF specifications are the creation and matching of large amounts of terms. Some of these terms may even be very big (more than 10^6 nodes). The amount of memory used during rewriting depends entirely on the number of terms being constructed and on the amount of storage each term occupies. In the case of innermost rewriting a lot of redundant (intermediate) terms are constructed.

At compile time, we can take various measures to avoid redundant term creation (only the last two have been implemented in the ASF+SDF compiler):

- Postponing term construction. Only the (sub)terms of the normal form must be constructed, *all* other (sub)terms are only needed to direct the rewriting process. By transforming the specification and extending it with rewrite rules that reflect the steering effect of the intermediate terms, the amount of term construction can be reduced. In the context of functional languages this technique is known as *deforestation* [Wad90]. Its benefits for term rewriting are not yet clear.
- Local sharing of terms, only those terms are shared that result from non-linear right-hand sides, e.g., `f(X) = g(X,X)`. Only those terms will be shared of which the sharing can be established at compile-time; the amount of sharing will thus be limited. This technique is also applied in ELAN [BKK$^+$96].
- Local reuse of terms, i.e., common subterms are only reduced once and their normal form is reused several times. Here again, the common subterm has to be determined at compile-time.

At run-time, there are various other mechanisms to reduce the amount of work:

- Storage of all original terms to be rewritten and their resulting normal forms, so that if the same term must be rewritten again its normal form is immediately available. The most obvious way of storing this information is by means of pairs consisting of the original term and the calculated normal form. However, even for small specifications and terms an explosion of pairs may occur. The amount of data to be manipulated makes this technique useless.

  A more feasible solution is to store only the results of functions that have been explicitly annotated by the user as "memo-function" (see Section 7.5).
- Dynamic sharing of (sub)terms. This is the primary technique we use and has already been discussed in Section 6.3.2.

7.3.1 Shared Terms versus Destructive Updates

Terms can be shared in a number of places at the same time, therefore they cannot be modified without causing unpredictable side-effects. This means that all operations on terms should be *functional* and that terms should effectively be *immutable* after creation.

During rewriting of terms by the generated code this restriction causes no problems since terms are created in a fully functional way. Normal forms are constructed bottom-up and there is no need to perform destructive updates on a term once it has been constructed. When normalizing an input term, this term is not modified, the normal form is constructed independent of the input term. If we would modify the input term we would get graph rewriting instead of (innermost) term rewriting. The term library is very general and is *not* only used for rewriting; destructive updates would therefore also cause unwanted side effects in other components based on this term library.

However, destructive operations on lists, like list concatenation and list slicing, become expensive. For instance, the most efficient way to concatenate two lists is to physically replace one of the lists by the concatenation result. In our case, this effect can only be achieved by taking the second list, prepending the elements of the first list to it, and return the new list as result.

In LISP, the success of hash consing [All78] has been limited by the existence of the functions `rplaca` and `rplacd` that can destructively modify a list structure. To support destructive updates, one has to support two kinds of list structures "mono copy" lists with maximal sharing and "multi copy" lists without maximal sharing. Before destructively changing a mono copy list, it has to be converted to a multi copy list. In the 1970's, E. Goto has experimented with a Lisp dialect (HLisp) supporting hash consing and list types as just sketched. See [TK90] for a recent overview of this work and its applications.

In the case of the ASF+SDF compiler, we *generate* the code that creates and manipulates terms and we can selectively generate code that copies subterms in cases where the effect of a destructive update is needed (as sketched above). This explains why we can apply the technique of subterm sharing with more success.

7.3.2 Reclaiming Unused Terms

During rewriting, a large number of terms is created, most of which will not appear in the end result. These terms are used as intermediate results to guide the rewriting process. This means that terms that are no longer used have to be reclaimed in some way.

After experimentation with various alternatives (reference counting, mark-and-compact garbage collection) we have finally opted for a mark–and-sweep garbage collection algorithm to reclaim unused terms. Mark-and-sweep collection is more efficient, both in time and space than reference counting [JL96]. The typical space overhead for a mark-sweep garbage collection algorithm is only 1 bit per object.

Mark-and-sweep garbage collection works using three (sometimes two) phases. In the first phase, all the objects on the heap are marked as 'dead'. In the second phase, all objects reachable from the known set of root objects are marked as 'live'. In the third phase, all 'dead' objects are swept into a list of free objects.

Mark-and-sweep garbage collection is also attractive, because it can be implemented efficiently in C and can work without support from the programmer or compiler [BW88]. We have implemented a specialized version of Boehm's conservative garbage collector [Boe93b] that exploits the fact that we are managing ATerms.

7.4 Benchmarks

Does maximal sharing of subterms lead to reductions in memory usage? How does it affect execution speed? Does the combination of techniques presented in this chapter indeed lead to an implementation of term rewriting that scales-up to industrial applications?

To answer these questions, we present in Section 7.4.1 three relatively simple benchmarks to compare our work with that of other efficient functional and algebraic language implementations. In Section 7.4.2 we give measurements for some larger ASF+SDF specifications.

7.4.1 Three Small Benchmarks

All three benchmarks are based on symbolic evaluation of expressions $2^n \mathbf{mod}\ 17$, with $17 \leq n \leq 23$. A nice aspect of these expressions is that there are many ways to calculate their value, giving ample opportunity to validate the programs in the benchmark. The actual source of the benchmarks can be obtained at

`http://www.wins.uva.nl/~olivierp/benchmark/index.html.`

Note that these benchmarks were primarily designed to evaluate *specific* implementation aspects such as the effect of sharing, lazy evaluation, and the like. They cannot (yet) be used to give an overall comparison between the various systems. Also note that some systems failed to compute results for the complete range $17 \leq n \leq 23$ in some benchmarks. In those cases, the corresponding graph also ends prematurely. Measurements were performed on an ULTRA SPARC-5 (270 MHz) with 256 Mb of memory. So far we have used the following implementations in our benchmarks:

- The ASF+SDF compiler as discussed in this chapter: we give results *with* and *without* maximal sharing.
- The Clean compiler developed at the University of Nijmegen [PE94]: we give results for standard (*lazy*) versions and for versions optimized with strictness annotations (*strict*).
- The ELAN compiler developed at INRIA, Nancy [BKK$^+$96].
- The Opal compiler developed at the Technische Universität Berlin [DFG$^+$94].
- The Glasgow Haskell compiler [JHH$^+$93].
- The Standard ML compiler [AM87].

The `evalsym` *Benchmark* The first benchmark is called `evalsym` and uses an algorithm that is CPU intensive, but does not use a lot of memory. This benchmark is a worst case for our implementation, because little can be gained by maximal sharing. The results are shown in Table 7.1. The differences between the various systems are indeed small. Although, ASF+SDF (with sharing) cannot benefit from maximal sharing, it does not loose much either.

Compiler	Time (sec)
Clean (strict)	32.3
SML	32.9
Clean (lazy)	36.9
ASF+SDF (with sharing)	37.7
Haskell	42.4
Opal	75.7
ASF+SDF (without sharing)	190.4
Elan	287.0

Table 7.1: The execution times for the evaluation of 2^{23}

The `evalexp` *Benchmark* The second benchmark is called `evalexp` and is based on an algorithm that uses a lot of memory when a typical eager (strict) implementation is used. Using a lazy implementation, the amount of memory needed is relatively small.

Memory usage is shown in Figure 7.9. Clearly, normal strict implementations cannot cope with the excessive memory requirements of this benchmark. Interestingly, ASF+SDF (with sharing) has no problems whatsoever due to the use of maximal sharing, although it is also based on strict evaluation

Execution times are plotted in Figure 7.10. Only Clean (lazy) is faster than ASF+SDF (with sharing) but the differences are small.

The `evaltree` *Benchmark* The third benchmark is called `evaltree` and is based on an algorithm that uses a lot of memory both with lazy and eager implementations. Figure 7.11 shows that neither the lazy nor the strict implementations can cope with the memory requirements of this benchmark. Only ASF+SDF (with sharing) can keep the memory requirements at an acceptable level due to its maximal sharing. The execution times plotted in Figure 7.12 show that only ASF+SDF scales-up for $n > 20$.

7.4.2 Compilation Times of Larger ASF+SDF Specifications

Table 7.2 gives an overview of the compilation times of four non-trivial ASF+SDF specifications and their sizes in number of equations, lines of ASF+SDF specification, and generated C code. The ASF+SDF compiler is the specification of the ASF+SDF to C compiler discussed in this chapter. The parser generator is an ASF+SDF specification which generates a parse table for an GLR-parser [Vis97]. The COBOL formatter is a pretty-printer for COBOL, this formatter is used within a renovation factory for COBOL [BSV97]. The Risla expander is an ASF+SDF specification of a domain-specific language for interest products, it expands modular Risla specifications into "flat" Risla specifications [ADR95]. These flat Risla spec-

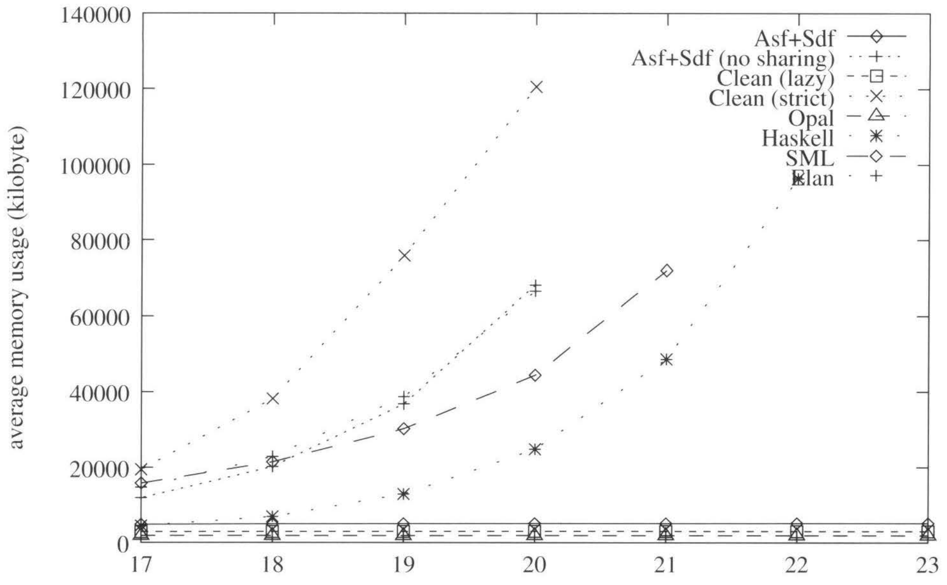


Figure 7.9: Memory usage for the `evalexp` benchmark

ifications are later compiled into COBOL code by an auxiliary tool. The compilation times in the column "ASF+SDF compiler" give the time needed to compile each ASF+SDF specification to C code. Note that the ASF+SDF compiler has been fully bootstrapped and is itself a compiled ASF+SDF specification. Therefore the times in this column give a general idea of the execution times that can be achieved with compiled ASF+SDF specifications. The compilation times in the last column are produced by a native C compiler (SUN's `cc`) with maximal optimizations.

Table 7.3 gives an impression of the effect of maximal sharing on execution time and memory usage of compiled ASF+SDF specifications. We show the results (with and without sharing) for the compilation of the ASF+SDF to C compiler itself and for the expansion of a non-trivial Risla specification.

7.5 Concluding Remarks

We have presented the techniques for the compilation of ASF+SDF to C, with emphasis on memory management issues. We conclude that compiled ASF+SDF specifications run with speeds comparable to that of other systems, while memory usage is in some cases an order of magnitude smaller. We have mostly used and adjusted existing techniques but their combination in the ASF+SDF compiler turns out to be very effective.

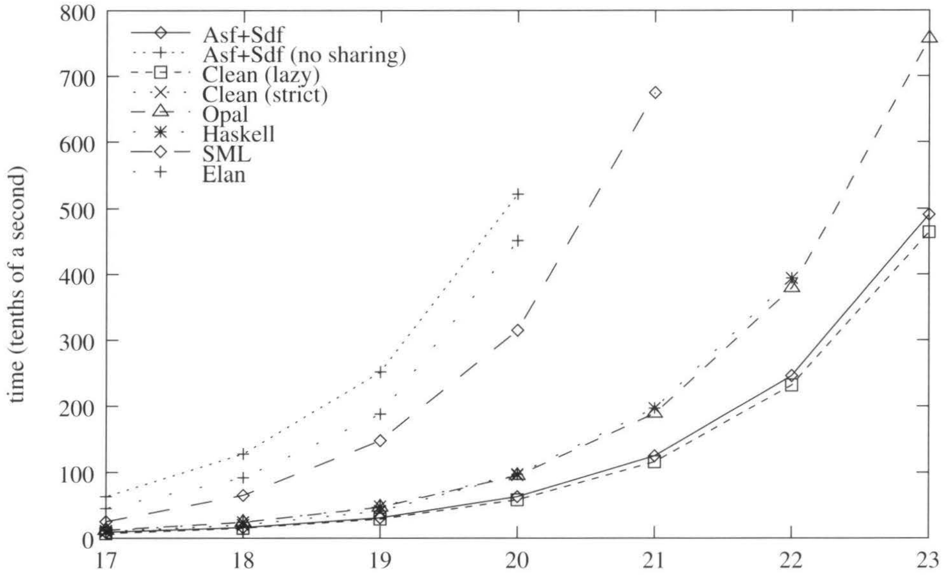


Figure 7.10: Execution times for the `evalexp` benchmark

It is striking that our benchmarks show results that seem to contradict previous observations in the context of SML [AG93] where sharing resulted in slightly increased execution speed and only marginal space savings. On closer inspection, we come to the conclusion that both methods for term sharing are different and can not be compared easily. We share terms immediately when they are created: the costs are a table lookup and the storage needed for the table while the benefits are space savings due to sharing and a fast equality test (one pointer comparison). In [AG93] sharing of subterms is *only* determined during garbage collection in order to minimize the overhead of a table lookup at term creation. This implies that local terms that have not yet survived one garbage collection are not yet shared thus loosing most of the benefits (space savings and fast equality test) as well. The different usage patterns of terms in SML and ASF+SDF may also contribute to these seemingly contradicting observations.

There are several topics that need further exploration. First, we want to study the potential of compile-time analysis for reducing the amount of garbage that is generated at run-time. Second, we have just started exploring the implementation of *memo-functions*. Although the idea of memo-functions is rather old, they have not be used very much in practice due to their considerable memory requirements. We believe that our setting of maximally shared subterms will provide a new perspective on the implementation of memo-functions. Finally, our ongoing concern is to achieve an even further scale-up of prototyping based on term rewriting.

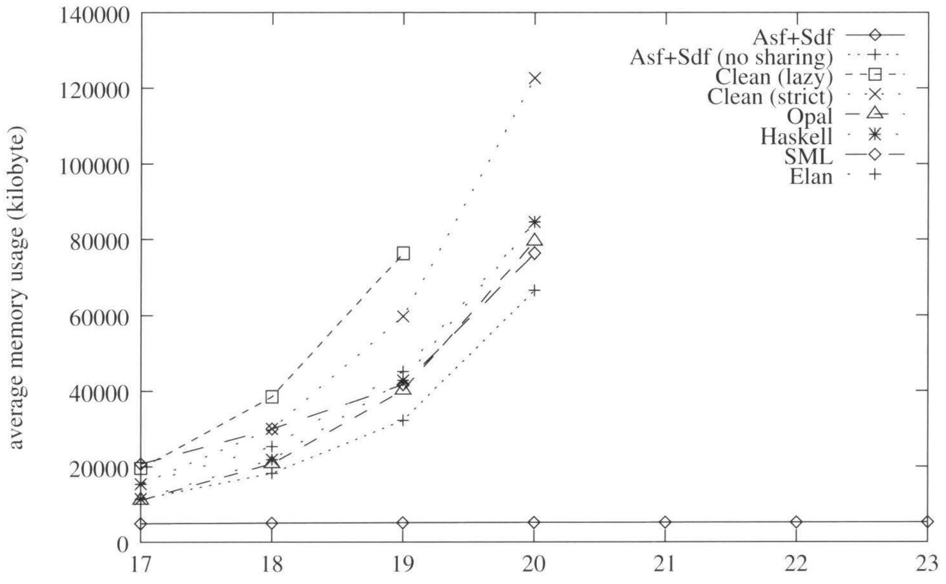


Figure 7.11: Memory usage for the `evaltree` benchmark

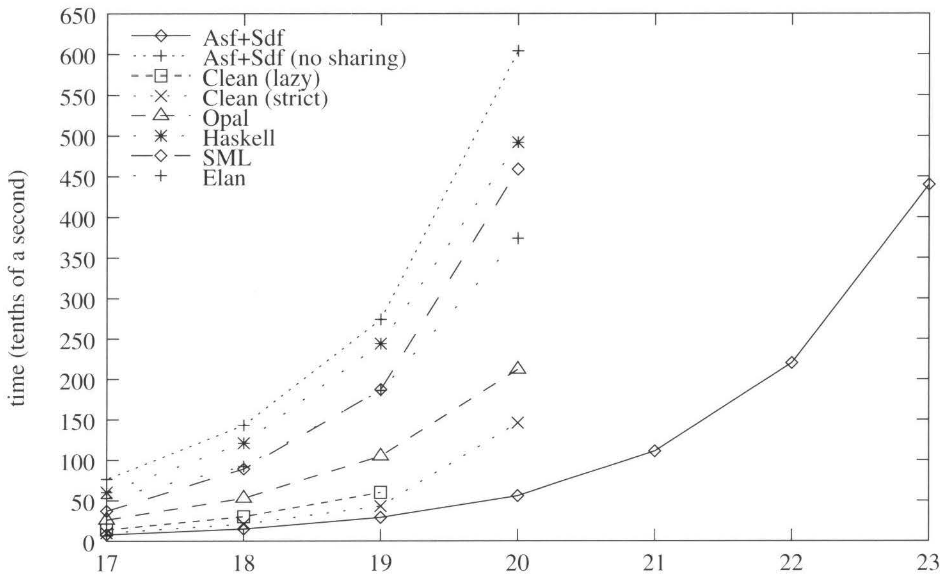


Figure 7.12: Execution times for the `evaltree` benchmark

Specification	ASF+SDF (equations)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF comp. (sec)	C comp. (sec)
ASF+SDF compiler	1876	8699	85185	216	323
Parser generator	1388	4722	47662	106	192
COBOL formatter	2037	9205	85976	208	374
Risla expander	1082	7169	46787	168	531

Table 7.2: Measurements of the ASF+SDF compiler

Application	Time (sec)	Memory (Mb)
ASF+SDF compiler (with sharing)	216	16
ASF+SDF compiler (without sharing)	661	117
Risla expansion (with sharing)	9	8
Risla expansion (without sharing)	18	13

Table 7.3: Performance with and without maximal sharing

8

Debugging ASF+SDF specifications

ASF+SDF specifications can be executed in two ways. By interpretation as discussed in Section 5.9, or by executing a compiled specification as discussed in Chapter 7. In this chapter, we will show how interpreted specifications can be debugged using TIDE. We will present some ideas on debugging compiled specifications, and we will show how TIDE support can be added to interpreters specified in ASF+SDF.

8.1 Introduction

The ASF+SDF Meta-Environment is a versatile system that can be used to develop executable specifications. An executable specification is nothing more than a computer program written in a formal language, so it is not surprising, that it is just as hard to write a fault-free specification as it is to write a fault-free program. Debugging support for developers of ASF+SDF specifications is therefore just as important as it is for any other software developer.

In this chapter, the two main themes of this thesis will come together. In the first part, we have presented a framework for 'generic debugging'. This framework significantly reduces the effort to build new debuggers. In the second part, we showed how ASF+SDF specifications can be executed, both using interpretation and compilation. In this last chapter, we will show how support for our generic debugging framework can be added to the interpreter discussed in Section 5.9. We will also give some ideas on how debugging support can be added to specifications compiled using the ASF+SDF compiler discussed in Chapter 7.

Because of the support for user-definable syntax, the ASF+SDF Meta-Environment is very well suited for the development of new programming languages. All kinds of tools can be generated for such specified languages, based on their specification in ASF+SDF. Typical tools include parsers, syntax-directed editors, pretty printers, and type checkers. It is also possible to specify tools that can be used to execute programs written in these specified languages: compilers and interpreters. But the possibility to execute these programs implies the need for debugging support at this level as well. To explore the possibilities in this area, this chapter contains a case study that shows how a compiler specified in ASF+SDF can be extended to instrument the code it generates with TIDE support.

8.2 Debugging Specifications

When designing TIDE support for a specific language, there are two important issues to resolve. The first is at which points during the execution of a program specific debug events have to be generated. The second issue is how to relate these events to the original source code using the `cpe` function discussed in Chapter 4.

To enable basic debugging support, three event ports are crucial: `stopped`, `started`, and `step`. The activation of `stopped` and `started` event rules is often straightforward. The real decisions involve when and how to activate event rules that have a `step` port, and which source coordinates should be returned by the `cpe` function at each activation of the `step` event rules.

In most imperative languages, `step` rules are either activated before or after execution of a single statement. The source code area that should be highlighted is most often the complete source line that contains the current instruction. In some cases this highlighting can be more precise, and the actual instruction being executed can be highlighted. Optionally, loops can generate more `step` events to provide a better understanding of what is going on, and to give the user more control over the execution. More `step` events might be generated at function entry or exit for the same purposes.

The semantics of ASF+SDF is based on innermost rewriting, as discussed in Chapter 7. This means that there is no notion of individual "statements" being executed, and so we cannot use the same debugging semantics as in the imperative case. In order to decide which semantics we want in this case, we have to take a close look at the different constructs that can appear in ASF+SDF specifications.

8.2.1 Unconditional Equations

Unconditional equations consist of a left-hand side and a right-hand side. The left-hand side consists of a term that might contain holes in the form of variables. With innermost rewriting, a term (or tree) is rewritten (or *reduced*) "inside out". This means that rewriting starts at the leaves of the tree that is being reduced. The current subterm being reduced is called the *redex*. The equations are tried one by one until the left-hand side of one of them matches the redex. In ASF+SDF, equations are tried in no particular order, except that special *default* equations are only tried when all other equations fail.

When a match is found, the matching equation will be *applied*. This means that the variables in the left-hand side of the matching equation are assigned a value corresponding to the subterms of the redex they matched with. Variables occurring in the right-hand side will be replaced by their values resulting in what is called the *reduct*. The redex in the original term will then be replaced by this reduct. Note that ASF+SDF only allows *bound* variables in the right-hand side of equations. This means that any variable that occurs in the right-hand side

```
module Naturals
imports Layout
exports
  sorts Nat
  context-free syntax
    zero           -> Nat
    succ(Nat)      -> Nat
    plus(Nat,Nat)  -> Nat

  variables
    [IJKL]         -> Nat

equations
  [n1] plus(zero, I) = I
  [n2] plus(succ(I),J) = succ(plus(I,J))
```

Figure 8.1: Specification of successor naturals

of an equation must be assigned a value using matching in the left-hand side of that equation, or in a condition as will be shown in the next subsection. A (sub-)term for which no equations are applicable is said to be in *normal form*.

Figure 8.1 shows an example of unconditional equations `n1` and `n2` that define the semantics of the `plus` operator over successor integers. Note that `zero` and `succ` have no equations defined over them and therefore they are called *constructors*, as they have no functionality of their own. They can only be used to construct and match terms. `plus` has two equations, `n1` and `n2`.

Suppose we want to reduce the term `plus(succ(zero), zero)`. As ASF+SDF rewrites left-most innermost, the left-most occurrence of `zero` is reduced first. As this is a constructor, it is already in normal form, and therefore left intact. Then `succ(zero)` is reduced, which is also left intact because `succ` is also a constructor. The second occurrence of zero is also in normal form. Then the complete term is reduced. This term does not match with the left-hand side of `n1`, but it does match with `n2`, so `n2` can be applied. During matching, `I` and `J` both are assigned the value `zero`. The redex (in this case the whole term) is replaced by the right-hand side of `n2`, after `I` and `J` are replaced by their values. This yields the term `succ(plus(zero,zero))`. At this point, the only term to which equations can be applied is `plus(zero,zero)`. Only `n1` matches with this subterm, resulting in the reduct `zero`. The normal form of the term now is `succ(zero)`.

8.2.2 Conditional Equations

ASF+SDF supports conditional equations where the conditions may be both positive and negative. This means that equation n2 in the previous example can be rewritten to:

```
[n2] I != zero,
     I = succ(K),
     L = succ(plus(K,J))
     ===================
     plus(I,J) = L
```

The first condition, I != zero is a *negative* condition. It only succeeds when I is not equal to zero. The second condition is a positive condition that introduces a new variable. The value of I is matched against succ(K), and if the match succeeds the subterm of I that matches with K is assigned to K. If the match does not succeed, the condition fails. When a side of a condition does not introduce new variables, it is first reduced before it is matched against the other side. This means that in the last condition, succ(plus(K,J)) is first reduced before its result is assigned to the new variable L. Note that the first condition is superfluous in this case, because the second condition also fails when I equals zero. Note also that the last condition is guaranteed to succeed, as the match between a term and a new variable is always successful.

8.2.3 List Matching

Yet another feature of ASF+SDF that has great impact on debugging is *list matching*. This powerful construct introduces the notion of *backtracking* in the semantics of ASF+SDF. Consider the example in Figure 8.2. This example specifies a list datatype with a "halve" operator that returns the first half of a list. The function "size" is not exported but only used to calculate the length of a list within this module..

Equation h1 is applicable for lists whose length is even. Equation h2 is applicable for lists whose length is odd. Both equations make essential use of backtracking. When the left-hand side of h1 matches, the list in the redex is split into two arbitrary sublists. The only condition of h1 than determines whether these parts are of equal length using the size function. If not, backtracking is used to find a different division of the list in the redex until all possible combinations are tried.

8.2.4 Stepping through Equations

Now we turn our attention to the issue of generating step events. In the previous examples a lot of individual steps were needed to reduce a simple term like plus(succ(zero), zero). It should be clear that if we generated a step event for each of these, stepping through all but the smallest reductions would take a long time. We need to keep the number

```
module Lists
imports Layout Naturals
exports
  sorts Elem List
  lexical syntax
    [a-z]                        -> Elem
  context-free syntax
    "[" { Elem "," }* "]"        -> List
    "halve" "(" List ")"         -> List
  variables
    "El" [\']*                   -> Elem
    "Els" [\']*                  -> { Elem "," }*

hiddens
  context-free syntax
    "size" "(" List ")"          -> Nat

equations
  [h1] size([Els]) = size([Els'])
       ==========================
       halve([Els,Els']) = [Els]

  [h2] size([Els]) = succ(size([Els']))
       ================================
       halve([Els,Els']) = [Els]

  [s1] size([El,Els]) = succ(size([Els]))

  [s2] size([]) = zero
```

Figure 8.2: Specification of a list datatype with a "halve" operator

of `step` events as low as possible, without loosing the ability to mentally trace the execution of our specification.

Unconditional Equations When applying an unconditional equation, two successive `step` events are generated. The first event is generated when the left-hand side of the equation is matched against the current redex and the match succeeds. In this case the left-hand side of the equation is considered as the current point of execution. The second step event is generated after the reduct is constructed, but before it is reduced. In this case, the right-hand side of the equation is considered as the current point of execution. When reducing the term `plus(succ(zero),zero)` as described in subsection 8.2.1, the generated `step` events can be used to produce the following highlight sequence:

1	$\boxed{\texttt{plus(succ(I),J)}}$ `= succ(plus(I,J))`	When `n2` matches with the input term, its left-hand side is highlighted.
2	`plus(succ(I),J) =` $\boxed{\texttt{succ(plus(I,J))}}$	The reduct is constructed, so the right-hand side of `n2` is highlighted.
3	$\boxed{\texttt{plus(zero,I)}}$ `= I`	`plus(zero,zero)` will be reduced, so the next `step` event is generated when the left-hand side of `[n1]` matches with this term.
4	`plus(zero,I) =` $\boxed{\texttt{I}}$	The last `step` event is generated when the reduct `zero` is returned.

Conditional Equations When debugging conditional equations, more `step` events need to be introduced to keep track of the execution path. Especially because evaluation of conditions might involve reduction of subterms. This may cause the debugger to display the equations involved in this "subreduction". It is crucial that the user receives concrete feedback after this subreduction is finished whether the condition that started the subreduction failed or succeeded.

We have identified three locations that need to generate `step` events to do this tracking:

- The first location is before reduction of the left-hand side of each condition. The complete left-hand side of the condition is considered as the current point of execution.
- The second location is before reduction of the right-hand side of each condition. The complete right-hand side of the condition is considered as the current point of execution.
- The third location is before checking whether the left-hand side and right-hand side of a condition match. The (in-)equality sign of the condition is considered as the current point of execution.

It can be argued that the last `step` event is not needed to keep track of the execution path. When after a subreduction a condition succeeds, the left-hand side of the next condition (or the right-hand side of the equation when no more conditions are present) is highlighted, so separate highlighting of the equality or inequality sign is not needed. A problem arises however when such a subreduction causes the condition to fail. If no more equations match with the current redex, a condition higher on the call stack could fail as well, transferring control to an equation several levels higher on the call stack.

Because this "equation hopping" can make it extremely difficult for the user to keep track of what is going on, we have decided to inform the user explicitly of success or failure of all conditions. Transitions between equations now only occur between equations one level below or above the current equation on the call stack.

To make the above discussion more comprehensible, we present the steps involved in reduction of the term `plus(succ(zero),zero)` using the second version of `n2`.

1	$\boxed{\texttt{plus(I,J)}}\ \texttt{= L}$	The left-hand side of `n2` is highlighted to indicate a successful match.
2	$\boxed{\texttt{I}}\ \texttt{!= zero,}$	The left-hand side of the first condition is highlighted.
3	$\texttt{I != }\boxed{\texttt{zero}}\texttt{,}$	The left-hand side does not need reducing, so the right-hand side is highlighted next.
4	$\texttt{I }\boxed{\texttt{!=}}\texttt{ zero,}$	The two sides of the condition are matched.
5	$\boxed{\texttt{I}}\ \texttt{= succ(K),}$	After this successful match, the left-hand side of the next condition is highlighted,
6	$\texttt{I = }\boxed{\texttt{succ(K)}}\texttt{,}$	followed by the right-hand side
7	$\texttt{I }\boxed{\texttt{=}}\texttt{ succ(K),}$	and the equality sign as the sides are matched.
8	$\boxed{\texttt{L}}\ \texttt{= succ(plus(K,J))}$	Now the left-hand side of the last condition of `n2` is highlighted.
9	$\texttt{L = }\boxed{\texttt{succ(plus(K,J))}}$	The right-hand side of the last condition is the only one that needs reducing. Before this reduction starts, the right-hand side is first highlighted. After this, we start reduction of the subterm `plus(K,J)`, where `K` and `J` are both equal to `zero`.
10	$\boxed{\texttt{plus(I,J)}}\ \texttt{= L}$	The left-hand side of both `n1` and `n2` match with the term `plus(zero,zero)`. In this example we assume that `n2` is tried first, so the left-hand side of `n2` is highlighted.
11	$\boxed{\texttt{I}}\ \texttt{!= zero,}$	Next the left-hand side of the first condition of `n2` is highlighted,
12	$\texttt{I != }\boxed{\texttt{zero}}\texttt{,}$	followed by its right-hand side
13	$\texttt{I }\boxed{\texttt{!=}}\texttt{ zero,}$	and the inequality sign.
14	$\boxed{\texttt{plus(zero,I)}}\ \texttt{= I}$	As the above condition fails, the `n1` is tried next to reduce `plus(zero,zero)`.
15	$\texttt{plus(zero,I) = }\boxed{\texttt{I}}$	This equation succeeds immediately, so the reduct `zero` is returned.
16	$\texttt{L }\boxed{\texttt{=}}\texttt{ succ(plus(K,J))}$	Now execution continues with the evaluation of the third condition of `n2` that started the reduction of `plus(zero,zero)` in step 9.

17	`plus(zero,I) =` $\boxed{\texttt{I}}$	Reduction now finishes and returns the normal form `succ(zero)`.

Debugging of Equations with List Matching. When an equation performs list matching, backtracking can occur when more than one match is possible. After the first match has been tried but this match caused a condition to fail, the next match is tried. Although the semantics of ASF+SDF do not predefine a particular order in which list matches are tried, most implementations will use a predefined ordering to simplify the implementation. The interpreter on which these examples are based orders the list matches by trying the smallest match first, from left to right. So, for instance, when we match the list pattern $L1, L2$ where $L1$ and $L2$ are list variables, against the list a, b, there are three possible matches which are tried in the following order:

1.	L1	=		L2	=	a,b
2.	L1	=	a	L2	=	b
3.	L1	=	a,b	L2	=	

By generating a single `step` event each time a new match is found, the user can keep track of the execution path. To demonstrate this, we will show the steps involved in reduction of the term `halve([a,b])` using the equations in Figure 8.2.

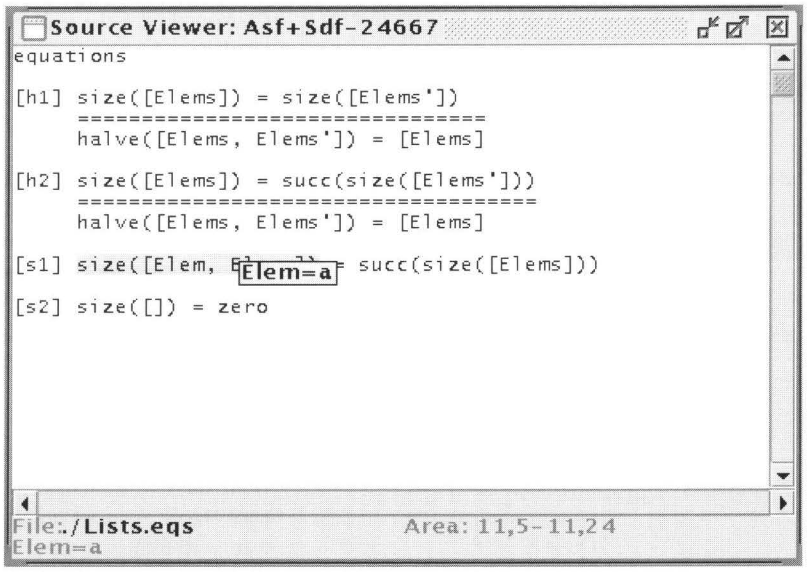
1	$\boxed{\texttt{halve([Els,Els'])}}$ `= [Els]`	The left-hand side of `h1` matches with the input term for the first time. The input list is split into two sublists. The empty list is assigned to `Els`, The rest of the list: `a,b` is assigned to `Els'`.
2	$\boxed{\texttt{size([Els])}}$ `= size([Els'])`	The left-hand side of the condition is reduced first. This involves reduction of `size([])`.
3	$\boxed{\texttt{size([])}}$ `= zero`	`s2` matches,
4	`size([]) =` $\boxed{\texttt{zero}}$	resulting in the reduct `zero`.
5	`size([Els]) =` $\boxed{\texttt{size([Els'])}}$	Now the right-hand side of the condition of `h1` is tried, which involves reducing `size([a,b])`.
6	$\boxed{\texttt{size([El,Els])}}$ `= succ(size([Els]))`	`s1` matches.
7	`size([El,Els]) =` $\boxed{\texttt{succ(size([Els]))}}$	Construction of the right-hand side of `s1` involves reduction of `size([b])`.
8	$\boxed{\texttt{size([El,Els])}}$ `= succ(size([Els]))`	Again `s1` is the only equation that matches.
9	`size([El,Els]) =` $\boxed{\texttt{succ(size([Els]))}}$	This time, the construction of the reduct involves reduction of `size([])`.

10	$\boxed{\texttt{size([])}}\ \texttt{= zero}$	Only `s2` matches.
11	$\texttt{size([Els]) = }\boxed{\texttt{size([Els'])}}$	The right-hand side of the condition of `h1` is also in normal-form.
12	$\texttt{size([Els]) }\boxed{\texttt{=}}\texttt{ size([Els'])}$	The left-hand side of the condition reduced to `zero`, the right-hand side to `succ(succ(zero))`, so the condition fails.
13	$\boxed{\texttt{halve([Els,Els'])}}\texttt{ = [Els]}$	At this point, the next match is tried. This time, the singleton list `a` is assigned to `Els`, and the singleton list `b` is assigned to `Els'`.
14	$\boxed{\texttt{size([Els])}}\texttt{ = size([Els'])}$	Evaluating the left-hand side of the condition of `h1` involves reduction of `size([a])`.
15	$\boxed{\texttt{size([El, Els])}}\texttt{ = succ(size([Els]))}$	The left-hand side of `s1` matches. `a` is assigned to `El`, and the empty list is assigned to `Els`.
16	$\texttt{size([El, Els]) = }\boxed{\texttt{succ(size([Els]))}}$	Construction of the right-hand side involves reduction of `size([])`.
17	$\boxed{\texttt{size([])}}\texttt{ = zero}$	`s2` matches.
18	$\texttt{size([]) =}\boxed{\text{zero}}$	`s2` is used to reduce `size([])` to `zero`.
19	$\texttt{size([Els]) = }\boxed{\texttt{size([Els'])}}$	Now the right-hand side of the condition of `h1` needs to be reduced.
20	$\boxed{\texttt{size([El,Els])}}\texttt{ = succ(size([Els]))}$	The left-hand side of `s1` matches. `b` is assigned to `El`, and the empty list is assigned to `Els`.
22	$\texttt{size([El,Els]) = }\boxed{\texttt{succ(size([Els]))}}$	The construction of the redex involves reduction of `size([])`.
22	$\boxed{\texttt{size([])}}\texttt{ = zero}$	`s2` matches.
23	$\texttt{size([]) = }\boxed{\texttt{zero}}$	`s2` is used to reduce `size([])` to `zero`.
24	$\texttt{size([Els]) }\boxed{\texttt{=}}\texttt{ size([Els'])}$	Both sides of the condition of `h1` have been reduced to a normal form, resulting in the condition `succ(zero) = succ(zero)` which succeeds.
25	$\texttt{halve([Els,Els']) = }\boxed{\texttt{[Els]}}$	At this point the reduct can be constructed: `[a]`, which represents the first half of the input list `[a,b]`.

8.2.5 TIDE Support in the Interpreter

The interpreter discussed in Section 5.9 can easily be instrumented to generate `step` events as discussed above. The only real problem is how to determine the source coordinates as returned by the `cpe` function. Fortunately, all layout information is still available in the AsFix representation of the equations (see Section 5.4.2). This allows us to *annotate* the AsFix representation of the equations with positional information before rewriting starts. All parts that are "interesting" for debugging purposes are annotated: left-hand side, right-hand side, and both sides of conditions. To store these annotations we use the generic ATerm annotation mechanism discussed in Chapter 6.

When an event rule is activated, the positional information is retrieved from the construct that caused the event rule to be fired. This construct can either be the left-hand side of the equation, the right-hand side, or the left-hand side or right-hand side of a condition. This positional information is then returned by invocations of the `cpe` function by the evaluation of the event condition or event actions. Figure 8.3 shows the TIDE support in the interpreter "in action".



```
equations

[h1] size([Elems]) = size([Elems'])
     ================================
     halve([Elems, Elems']) = [Elems]

[h2] size([Elems]) = succ(size([Elems']))
     ====================================
     halve([Elems, Elems']) = [Elems]

[s1] size([Elem, E Elem=a = succ(size([Elems]))

[s2] size([]) = zero
```

```
File:./Lists.eqs                    Area: 11,5-11,24
Elem=a
```

Figure 8.3: The List example in TIDE

As always in TIDE, the user can right-click on a variable to view its current value.

8.2.6 TIDE Support in the Compiler

So far we have shown how we implemented basic debugging support for the ASF+SDF interpreter. We would also like to implement debugging support for specifications compiled with the ASF+SDF compiler presented in Chapter 7. In this section we will discuss two possible approaches to reach this goal.

Generating instrumented code The first approach is to modify the compiler to generate code that is instrumented with extra debugging statements. Every code chunk that might give rise to the activation of `step` events can be instrumented with calls to the debugger.

This approach suffers from a problem that is common to most compilers: optimizations performed during compilation can influence the debugging semantics. In the case of the ASF+SDF compiler, identical left-hand sides and conditions of different equations can potentially result in generation of a single chunk of code. We cannot relate such a chunk back to a single location in the source code, so debugging is hampered. It is therefore important to disable this kind of optimizations when generating debug code.

Specification transformation A much cleaner but potentially less efficient approach is to implement the debugging of compiled specifications using *transformation* of the original specification. Figure 8.4 shows a transformed version of the equations of the naturals example, based on the conditional version of `[n2]`. Note that each call to the `step` function contains the filename and coordinates of the corresponding area in the original source code. The coordinates consist of two pairs of integers, the first pair supplying the line and column of the start of the area, and the second one representing the line and column of the end of the area.

The syntax of the `step` function and its argument are specified in a special module `Debug` that is imported by the transformed module. The `Debug` module also contains a default definition of the `step` function:

```
[step] step(Info) = true
```

Basically, for each position that could possibly generate a `step` event, a new condition is introduced. These new conditions are guaranteed to succeed, as the `step` function always returns `true`. This means that the semantics of the original specification are kept intact. The calls to the `step` function takes the positional information of the `step` event as an argument.

Without any support from the run time system, the transformed specification behaves exactly like the original specification. When the run time system is augmented with debugging support, the invocation of the `step` function is caught, so the appropriate `step` events can be activated.

At first glance, the transformational approach looks very clean. No changes to the compiler are needed, only a small extension of the run time system is required. There are some disadvantages however. As is clear from this small example, the amount of extra code that

```
equations
  [n1] step("Naturals:2,7-2,20") = true,
       step("Naturals:2,23-2,24") = true
       ====================================
       plus(zero, I) = I

  [n2] step("Naturals:8,7-8,17") = true,
       step("Naturals:4,7-4,8") = true,
       X0 = zero,
       step("Naturals:4,12-4,16") = true,
       I != X,
       step("Naturals:5,7-5,8") = true,
       X1 = succ(K),
       step("Naturals:5,11-5,18") = true,
       I = X1,
       step("Naturals:6,7-6,8") = true,
       X2 = succ(plus(K,J)),
       step("Naturals:6,11-6,26") = true,
       L = X2
       step("Naturals:7,19-7,20") = true
       ===================
       plus(I,J) = L
```

Figure 8.4: A transformed version of the successor naturals

needs to be generated is substantial. In general, the size of the specification is increased by a factor 2.

In [Alb97] the transformational approach to debugging ASF+SDF specifications is pursued further. This work shows that if we add more debug features, the complexity of this approach increases dramatically. For instance, it is shown that it is possible to add transformations that enable the debugger to access the value of variables. In order to do this extra arguments need to be generated for the `step` function. But to do this, we also need to generate extra syntax rules to inject the sort of each variable into one "super sort" so variables can be passed to the step function in a type-safe manner.

8.3 Adding Debugging Support to a Specified Compiler

The ASF+SDF Meta-Environment is first and foremost a tool to specify tools for programming languages. Some of these tools like compilers and interpreters are primarily aimed at executing programs written in specified languages. But if we are able to execute programs using these specified tools, a natural need arises to debug these programs as well. In this section we will present a case study that uses TIDE for this purpose. We show how an existing compiler for a toy language called PICO can be extended in such a way that this compiler is capable of generating code augmented with TIDE support. In this way, TIDE provides us with a fully featured interactive debugger for PICO programs at a fraction of the cost it would take to develop a fully featured PICO debugger from scratch.

8.3.1 The PICO to C Compiler

PICO is a small Pascal-like language whose key syntax elements can be written down in about two dozen lines of SDF, as shown in Figure 8.5. Note that a number of modules are imported which are not shown here. These modules define the syntax of identifiers, integers and string constants.

Figure 8.6 shows an example of a PICO program that calculates the factorial of the number four. We will use this example throughout this section to explain the code generation of the PICO to C compiler and the generation of debug code.

The PICO to C compiler compiles the factorial PICO program into a single C function shown in Figure 8.7. This function is compiled together with a small run time library for PICO (less than 150 lines of code) yielding an executable program. The run time library defines the `PICONATURAL` macro that is used to register variables, and the `finish` function that prints the value of all variables at the end of the program. It is also responsible for providing a `main` function that calls the actual `picomain` function after doing some initialization.

When we run the resulting executable we get the following output:

```
~/Research/pico/non-debug> ./fac
natural input = 1
natural output = 24
natural repnr = 1
natural rep = 12
~/Research/pico/non-debug>
```

which is a list of all variables and their values at the end of the run.

8.3.2 Adding TIDE Support

To add TIDE we need to do two things. Extra debug statements need to be inserted in the generated C code, and the run time library needs to be extended to handle these extra debug statements and to make the value of variables available to TIDE.

```
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
  sorts PROGRAM DECLS ID-TYPE STATEMENT EXP
  context-free syntax
    "begin" DECLS {STATEMENT ";"}* "end"    -> PROGRAM
    "declare" {ID-TYPE  ","}* ";"           -> DECLS
    PICO-ID ":" TYPE                        -> ID-TYPE
    PICO-ID ":=" EXP                        -> STATEMENT
    "if" EXP "then" {STATEMENT ";"}*
        "else" {STATEMENT ";"}* "fi"        -> STATEMENT
    "while" EXP "do" {STATEMENT ";"}* "od" -> STATEMENT
    PICO-ID                                 -> EXP
    PICO-NAT-CON                            -> EXP
    PICO-STR-CON                            -> EXP
    EXP "+" EXP                             -> EXP {left}
    EXP "-" EXP                             -> EXP {left}
    EXP "||" EXP                            -> EXP {left}
    "(" EXP ")"                             -> EXP {bracket}
```

Figure 8.5: The syntax of PICO

Figure 8.8 shows what the generated code looks like when debug statements are inserted at compile time. Close inspection of this code reveals that it is exactly the same as the code in Figure 8.7, except that one `debugstep(xx)` statement has been added for each PICO statement in the original PICO program. Moreover, the number *xx* corresponds to the line number of the original PICO statement.

At first glance, it might seem straightforward to insert these extra `debugstep` statements, but unfortunately there is a problem. The current version of ASF+SDF in which the PICO to C compiler has been implemented does not make layout information available to the specification writer. This means that it is not possible to determine the current line number during compilation, so it is impossible to insert the correct line numbers in the `debugstep` calls. We circumvented this problem by first pre-processing the original PICO program before compiling it. During this pre-processing stage we simply insert a '!' character at the start of every line. This makes it possible for the compiler to count '!' characters to determine the current line number.

Unfortunately this solution implies that the original specification of the PICO to C compiler has to change drastically before it can generate debug information. First of all, the syntax

```
begin
  declare
    input : natural,
    output  :  natural,
    repnr: natural,
    rep: natural;

  input := 4;
  output := 1;
  while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
      output := output + rep;

      repnr := repnr - 1
    od;
    input := input - 1
  od
end
```

Figure 8.6: Calculation the factorial of four in PICO

definition of PICO itself has to be adapted to allow the '!' characters, as shown in Figure 8.9. Note that we do not allow newlines in expressions to simplify the compiler specification.

In addition the actual compiler specification needs to be adapted to cope with the appearance of '!' characters, as well as to insert `debugstep` calls at the correct places. This results in a 50% increase in the size of the compiler, most of which is due to the extra complexity of keeping track of the current line number.

The PICO run time library is extended with `debugstep` function, in such a way that calls to this function result in `step` events in TIDE. In addition, each call to `debugstep` also generates a `location` event to support breakpoints in PICO programs. The run time library is also extended to support the `var` function in debug actions and conditions to make the value of variables available to TIDE. These extensions to the PICO run time library are implemented using only 120 lines of code.

This case study shows that adding debugging support to an existing compiler specified in ASF+SDF can be straightforward but for one problem: keeping track of positional information during compilation. This problem can be solved by preprocessing the original source

```
void picomain (  )
{
  PICONATURAL ( input ) ;
  PICONATURAL ( output ) ;
  PICONATURAL ( repnr ) ;
  PICONATURAL ( rep ) ;
  input = 4 ;
  output = 1 ;
  while ( input - 1 ) {
    rep = output ;
    repnr = input ;
    while ( repnr - 1 ) {
      output = output + rep ;
      repnr = repnr - 1 ;
    } input = input - 1 ;
  }
  finish();
}
```

Figure 8.7: The factorial example compiled to C code

code, but this severely increases the complexity of the original compiler specification. Ideally, this positional information should be accessible directly from an ASF+SDF specification, for instance by exposing layout information at the specification level.

After tackling this problem, it takes a minimal amount of work to connect a language run time system to TIDE, immediately yielding a fully functional debug implementation. Figure 8.10 shows a screenshot of a PICO debugging session using TIDE. It is clear that the conventional method of developing a new debugger for a programming language would require considerably more human resources than are needed using the TIDE approach.

8.4 Related Work

In [Deu94, Tip95, DKT96] a technique called *origin tracking* is used to generate language-specific debugging tools for algebraic specifications. In origin tracking, at each reduction step from input term to normal form relations are maintained between subterms in the redex and subterms in the reduct. When combined properly, these relations can for instance be used to find the relation between PICO statements and corresponding C statements in the generated C program in our case study. It will be interesting to see if we can combine origin tracking with

```
void picomain (  )
{
  PICONATURAL ( input );
  PICONATURAL ( output );
  PICONATURAL ( repnr );
  PICONATURAL ( rep );
  debugstep(8);
  input = 4;
  debugstep(9);
  output = 1;
  debugstep (10);
  while ( input - 1 ) {
    debugstep(11);
    rep = output;
    debugstep(12);
    repnr = input;
    debugstep(13);
    while ( repnr - 1 ) {
      debugstep(14);
      output = output + rep;
      debugstep(16);
      repnr = repnr - 1;
    }
    debugstep(18);
    input = input - 1;
  }
  finish();
}
```

Figure 8.8: The factorial example compiled to C code with debug information

our techniques to derive TIDE support automatically without the need to adapt the original compiler specification.

Another example of a system that is capable of generating debugging tools is described in [BMS87]. In this work specifications of the denotational semantics of a programming language are compiled into a functional language. Debugger behavior can be expressed using a set of builtin debugging concepts, not unlike TIDE. Examples of these notions are trace functions, breakpoint definitions and state inspection primitives.

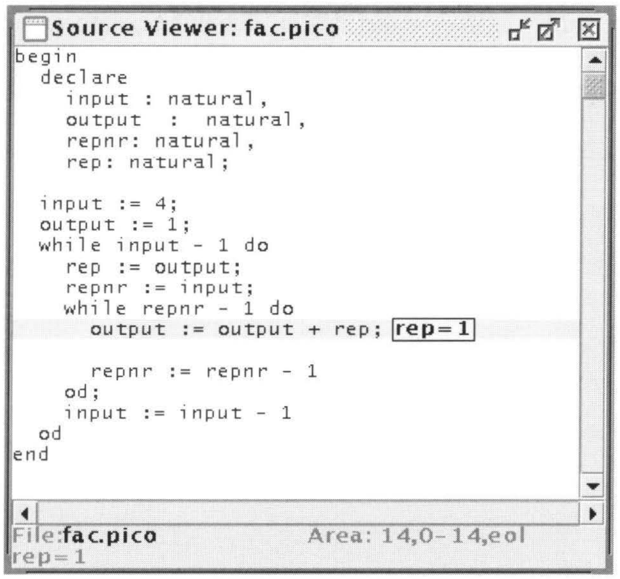
```
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
  sorts PROGRAM DECLS ID-TYPE STATEMENT EXP SOL L LSTAT
  lexical syntax
    "!"        -> SOL
  context-free syntax
    SOL*                                       -> L
    L "begin" DECLS { LSTAT ";" }* L "end" -> PROGRAM
    L STATEMENT                                -> LSTAT
    L "declare" {ID-TYPE  ","}* ";"            -> DECLS
    L PICO-ID ":" TYPE                         -> ID-TYPE
    PICO-ID ":=" EXP                           -> STATEMENT
    "if" EXP L "then" {LSTAT ";"}* L
       "else" {LSTAT ";"}* L "fi"              -> STATEMENT
    "while" EXP L "do" {LSTAT ";"}* L "od" -> STATEMENT
    PICO-ID                                    -> EXP
    PICO-NAT-CON                               -> EXP
    PICO-STR-CON                               -> EXP
    EXP "+" EXP                                -> EXP {left}
    EXP "-" EXP                                -> EXP {left}
    EXP "||" EXP                               -> EXP {left}
    "(" EXP ")"                                -> EXP {bracket}
```

Figure 8.9: The syntax of PICO where each line can start with a '!' character. Note that the occurence of line breaks is restricted to specific positions. For instance, a single expression cannot be spread over several lines.

In [Ber91] the operational semantics of a programming language are extended with *semantic display rules* to generate *animators*. Different views on the execution of a program can be generated using different display rules. Static views are based on the abstract syntax tree of a program, dynamic views are based on the program state during execution.

8.5 Conclusions

We have added TIDE support to the ASF+SDF Meta-Environment on multiple levels. TIDE is used to create a debugger for ASF+SDF specifications. We also showed that it is possible to create a debugger for specified languages. In all these cases, the effort needed to build a



```
begin
  declare
    input : natural,
    output  :  natural,
    repnr: natural,
    rep: natural;

  input := 4;
  output := 1;
  while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
      output := output + rep; rep=1

      repnr := repnr - 1
    od;
    input := input - 1
  od
end
```

File:fac.pico Area: 14,0-14,eol
rep=1

Figure 8.10: Debugging a PICO program using TIDE

Component	Implementation language	Original size (lines of code)	Tide support (lines of code)
ASF+SDF interpreter	C	3445	185†
Pico2C compiler	ASF+SDF	465	30
Pico2C run time support	C	183	105†

†: In addition a C library is used that implements generic TIDE debug-adapter support. This library consists of 900 lines of C code and can be reused for every debug-adapter written in C (see also Figure 4.4).

Figure 8.11: Code increase when TIDE support is added

sophisticated debugger based on TIDE is negligible compared to the effort it would take to develop a debugger from scratch. This claim is supported by the statistics in Figure 8.11. This figure gives an impression of the amount of source code that is needed to add TIDE support to the components discussed in this chapter.

These figures show that the code needed to actually add TIDE support to a system can be very small. If we compare the two run time systems (ASF+SDF interpreter and the Pico2C run time support), it is interesting to notice that both need about the same amount of extra code to support the same TIDE features, even though the ASF+SDF interpreter run time system is vastly more complex than the Pico2C run time system.

On the negative side, we observe that lack of line number information on the level of ASF+SDF specifications complicates the addition of TIDE support in specified compilers. Given the fact that we have used TIDE to build realistic debuggers for languages like C and Java, we speculate that adding TIDE support to compilers specified in ASF+SDF is within reach once the line number problem has been solved.

9
Concluding remarks

In Chapter 1 we presented three central research questions. Reaching the end of this thesis it is time to reflect on these questions and to draw some conclusions.

9.1 Feasibility of Generic Debugging

Research Question 1: *Is it possible to develop generic debugging technology that can be used to significantly reduce the cost of developing debuggers for new languages?*

The first part of this thesis shows that for features found in modern debuggers it is possible to abstract from the actual semantics of the language under consideration. Based on this approach we have built a debugger implementation called TIDE. Because of the generic model on which TIDE is based, developing debug support for new programming languages is inexpensive because most of the TIDE implementation can be reused. Another advantage is that users of TIDE are not forced to use yet another debugger when switching programming languages. This is a major asset, especially because component architectures are growing in popularity, allowing distributed applications to become more and more heterogeneous.

We have demonstrated the feasibility of generic debugging using a relatively small set of features including single stepping, conditional breakpoints, and inspection of the value of variables. Features like static scoping and inspection of stack frames are only briefly discussed and features like inspection of large datastructures and post-mortem debugging are completely ignored.

The programming languages considered cover a wide range of paradigms: debugging support has been presented for a concurrent language (ToolBus script), as well as an object oriented language (Java), several imperative languages (C, Tcl) and a language whose semantics is based on term rewriting (ASF+SDF).

In both areas (features and languages) more work is needed to extend the range of debugging activities supported by our framework.

9.2 Usability of Maximal Term Sharing

Research Question 2: *Can maximal term sharing be used to increase both the time and space efficiency of executable algebraic specifications?*

In Chapter 6 we have presented a design and implementation of the ATerm datatype that makes use of maximal sharing. In Chapter 7 we have shown that this implementation provides an excellent foundation to build the runtime environment of the ASF+SDF to C compiler. Using three benchmarks we have shown that our compiler compares favorably to other mainstream compilers for functional and algebraic languages. The use of maximal sharing results directly in a dramatic decrease in memory consumption, especially when terms contain much redundant information. The performance penalty associated with maintaining maximum sharing is offset by a couple of performance gains. The reduced memory usage decreases the cost of garbage collection and decreases cache and page misses. On top of this, deep equality checking of terms can be replaced by pointer equality checking because terms are only equal when they are in fact the same term.

9.3 Debugging in the Context of ASF+SDF

Research Question 3: *Can generic debugging technology be used in the* ASF+SDF *Meta-Environment at the following three levels: Debugging of the* ASF+SDF *Meta-Environment itself, debugging of* ASF+SDF *specifications, and debugging of programs written in languages specified in* ASF+SDF*?*

We can answer this question by studying the three levels of debugging separately:

- *Debugging the* ASF+SDF *Meta-Environment itself*: The possibilities in this area are implicitly covered in the first part of this thesis. The components of the ASF+SDF Meta-Environment are written in the following programming languages: C, ASF+SDF, Java, and Tcl. As TIDE support is available for all of these languages, as well as for the TOOLBUS scripts that form the communication backbone of the ASF+SDF Meta-Environment, TIDE can be used to debug the ASF+SDF Meta-Environment itself.
- *Debugging* ASF+SDF *specifications*: In Section 8.2 TIDE support for the ASF+SDF interpreter is presented in detail, including the mapping of the semantic features of ASF+SDF onto TIDE primitives. The result is a useful debugging system for ASF+SDF specifications. Some ideas for instrumenting the code generated by the ASF+SDF compiler were also discussed.
- *Debugging of programs written in languages specified in* ASF+SDF: In Section 8.3 we have presented a case study that showed a possible approach for this problem in the case

of a specified compiler for PICO. This case study encountered a number of problems, the most severe one is that it is very difficult to keep track of positional information. Extensions to both ASF+SDF and tools for evaluating ASF+SDF specifications could alleviate this problem considerably.

A
Syntax of TOOLBUS Scripts

In this appendix we will give the complete definition of the syntax of TOOLBUS scripts using SDF (Syntax Definition Formalism, see [HHKR92]) which is part of the ASF+SDF formalism that has been discussed in the second part of this thesis. SDF should be quite easy to understand for people who are familiar with syntax formalisms like BNF. The main difference is that in SDF the non-terminals (sorts) are placed on the right-hand side instead of on the left.

```
exports
  sorts BOOL NAT INT SIGN EXP UNSIGNED-REAL REAL STRING
        ID NAME VNAME BSTR TERM TERM-LIST VAR GEN-VAR TYPE
        ATOM ATOMIC-FUN PROC PROC-APPL FORMALS TIMER-FUN
        FEATURE-ASG FEATURES TB-CONFIG DEF T-SCRIPT
  lexical syntax
        [ \t\n]                               -> LAYOUT
        "%%" ~[\n]*                           -> LAYOUT

        [0-9]+                                -> NAT
        NAT                                   -> INT
        SIGN NAT                              -> INT
        [+\-]                                 -> SIGN

        [eE] NAT                              -> EXP
        [eE] SIGN NAT                         -> EXP
        NAT "." NAT                           -> UNSIGNED-REAL
        NAT "." NAT EXP                       -> UNSIGNED-REAL
        UNSIGNED-REAL                         -> REAL
        SIGN UNSIGNED-REAL                    -> REAL

        [a-z][A-Za-z0-9\-]*                   -> ID
        "\"" ~[\"]* "\""                      -> STRING
        [A-Z][A-Za-z0-9\-]*                   -> NAME
        [A-Z][A-Za-z0-9\-]*                   -> VNAME
        [a-z][a-z\-]*                         -> ATOMIC-FUN
```

```
      delay                              -> TIMER-FUN
      abs-delay                          -> TIMER-FUN
      timeout                            -> TIMER-FUN
      abs-timeout                        -> TIMER-FUN
context-free syntax
      true                               -> BOOL
      false                              -> BOOL
      BOOL                               -> TERM
      INT                                -> TERM
      REAL                               -> TERM
      STRING                             -> TERM

      TERM                               -> TYPE

      VNAME                              -> VAR
      VNAME ":" TYPE                     -> VAR

      VAR                                -> GEN-VAR
      VAR "?"                            -> GEN-VAR
      GEN-VAR                            -> TERM
      "<" TERM ">"                       -> TERM
      ID                                 -> TERM
      ID "(" TERM-LIST ")"               -> TERM
      {TERM ","}*                        -> TERM-LIST
      "[" TERM-LIST "]"                  -> TERM

      NAME                               -> VNAME

      ATOMIC-FUN "(" TERM-LIST ")"       -> ATOM
      delta                              -> ATOM
      tau                                -> ATOM
      create "(" PROC-APPL ","  TERM ")" -> ATOM
      ATOM TIMER-FUN "(" TERM ")"        -> ATOM
      VNAME ":=" TERM                    -> ATOM

      ATOM                               -> PROC
      PROC "+" PROC                      -> PROC  {left}
      PROC "." PROC                      -> PROC  {right}
      PROC "||" PROC                     -> PROC  {right}
      PROC "*" PROC                      -> PROC  {left}
      "(" PROC ")"                       -> PROC  {bracket}
      if TERM then PROC else PROC fi     -> PROC
      if TERM then PROC fi               -> PROC
      execute(TERM-LIST)                 -> PROC
      let {VAR ","}* in PROC endlet      -> PROC
```

```
     NAME                                 -> PROC-APPL
     NAME "(" TERM-LIST ")"               -> PROC-APPL
     PROC-APPL                            -> PROC

     "(" {GEN-VAR ","}* ")"               -> FORMALS
                                          -> FORMALS

     process NAME FORMALS is PROC         -> DEF
     ID "=" STRING                        -> FEATURE-ASG
    "{" { FEATURE-ASG  ";"}* "}"          -> FEATURES
     tool ID FORMALS is FEATURES          -> DEF
     toolbus "("{PROC-APPL ","}+ ")"      -> TB-CONFIG
     DEF* TB-CONFIG                       -> T-SCRIPT

priorities
    PROC "*" PROC -> PROC > PROC "." PROC -> PROC >
    PROC "+" PROC -> PROC > PROC "||" PROC -> PROC
```

B
TOOLBUS Primitives

Primitive	Description
`delta`	inaction (deadlock)
`tau`	internal step
P_1`+`P_2	choice
P_1`.`P_2	sequential composition
P_1`||`P_2	parallel composition
P_1`*`P_2	iteration
`if` T `then` P `fi`	guarded command
`if` T `then` P_1 `else` P_2 `fi`	conditional
`create(`Pnm`(`T`, ...), `Pid`?)`	process creation[1]
V `:=` T	assignment, T expression
`snd-msg(`T`,...)`	send a message (binary, synchronous)
`rec-msg(`T`,...)`	receive a message (binary, synchronous)
`snd-note(`T`)`	send a note (broadcast, asynchronous)
`rec-note(`T`)`	receive a note (asynchronous)
`no-note(`T`)`	no notes available for process
`subscribe(`T`)`	subscribe to notes
`unsubscribe(`T`)`	unsubscribe from notes
`delay(`T`)`	relative time delay of atom
`abs-delay(`T`,...)`	absolute time delay of atom[2]
`timeout(`T`)`	relative timeout of atom
`abs-timeout(`T`,...)`	absolute timeout of atom[2]
`rec-connect(`Tid`?)`	receive a connection request from a tool
`rec-disconnect(`Tid`?)`	receive a disconnection request form a tool
`execute(`Tnm`(`T`,...), `Tid`?)`	execute a tool[1]
`snd-terminate(`Tid`, `T`)`	terminate the execution of a tool
`snd-eval(`Tid`, `T`)`	send evaluation request to tool
`snd-cancel(`Tid`)`	cancel an evaluation request to tool†
`rec-value(`Tid`, `T`)`	receive a value from a tool
`snd-do(`Tid`, `T`)`	send request to tool (no return value)
`rec-event(`Tid`, `T`, ...)`	receive event from tool
`snd-ack-event(`Tid`, `T`)`	acknowledge a previous event from a tool

Primitive	Description
`shutdown(`T`)`	terminate TOOLBUS
`reconfigure`	reconfigure TOOLBUS†
`attach-monitor`	attach a monitoring tool to a process†
`detach-monitor`	detach a monitoring tool from a process†
`printf(`S`, `T`, ...)`	print terms (after variable replacement) according to format S
`read(`T_1`, `T_2`)`	give prompt T_1, read term, should match with T_2
`process` Pnm`(`F`, ...) is` P	process definition[3]
`let` F`, ... in` P `endlet`	declare variables in P
`tool` Tnm`(`F`, ...) is {`$Feat$`, ...}`	tool definition[3]
`host =` Str	host feature in tool definition
`command =` Str	command feature in tool definition
`details = <<` $Lines$ `>>`	details feature in tool definition
`toolbus(`Pnm`(`T`,...), ...)`	TOOLBUS configuration

Notes

1	$(T, \ldots)$ is optional
2	Absolute time described by a 6-tuple ($year$, $month$, day, $hour$, $minutes$, $seconds$), with $year \geq 95$, $1 \leq month \leq 12$, $1 \leq day \leq 31$, $0 \leq hour \leq 23$, $0 \leq minutes \leq 59$, and $0 \leq seconds \leq 61$ (seconds can be greater than 59 to allow leap seconds). Absolute time may be abbreviated, by omitting, at most, the first three elements of the 6-tuple. Omitted elements default to their current value.
3	$(F, \ldots)$ is optional
†	Not yet implemented

Legendum

T	term
$T, \ldots$	list of terms separated by comma's
V	variable
F	declaration of formal or local variable of the form $V : Type$
P, P_1, P_2	process expression
Tid	tool identifier, a variable of type Tnm (with Tnm declared as tool name)
Tnm	tool name
Pnm	process name
Pid	process identifier, a variable of type `int`
Str	a string constant
$Lines$	list of lines

C

TOOLBUS Functions

C.1 Boolean functions

Function	Result type	Description
`not(<bool>`$_1$`)`	`<bool>`	$\neg$ `<bool>`$_1$
`and(<bool>`$_1$`,<bool>`$_2$`)`	`<bool>`	`<bool>`$_1$ $\wedge$ `<bool>`$_2$
`or(<bool>`$_1$`,<bool>`$_2$`)`	`<bool>`	`<bool>`$_1$ $\vee$ `<bool>`$_2$
`equal(<term>`$_1$`, <term>`$_2$`)`	`<bool>`	`<term>`$_1$ $\equiv$ `<term>`$_2$; for lists multi-set equality
`not-equal(<term>`$_1$`, <term>`$_2$`)`	`<bool>`	`not(equal(<term>`$_1$`, <term>`$_2$`))`
`less(<int>`$_1$`,<int>`$_2$`)`	`<bool>`	`<int>`$_1$ $<$ `<int>`$_2$
`less-equal(<int>`$_1$`,<int>`$_2$`)`	`<bool>`	`<int>`$_1$ $\leq$ `<int>`$_2$
`greater(<int>`$_1$`,<int>`$_2$`)`	`<bool>`	`<int>`$_1$ $>$ `<int>`$_2$
`greater-equal(<int>`$_1$`,<int>`$_2$`)`	`<bool>`	`<int>`$_1$ $\geq$ `<int>`$_2$
`rless(<real>`$_1$`,<real>`$_2$`)`	`<bool>`	`<real>`$_1$ $<$ `<real>`$_2$
`rless-equal(<real>`$_1$`,<real>`$_2$`)`	`<bool>`	`<real>`$_1$ $\leq$ `<real>`$_2$
`rgreater(<real>`$_1$`,<real>`$_2$`)`	`<bool>`	`<real>`$_1$ $>$ `<real>`$_2$
`rgreater-equal(<real>`$_1$`,<real>`$_2$`)`	`<bool>`	`<real>`$_1$ $\geq$ `<real>`$_2$

C.2 Arithmetic functions

`add(<int>`$_1$`,<int>`$_2$`)`	`<int>`	`<int>`$_1$ + `<int>`$_2$
`sub(<int>`$_1$`,<int>`$_2$`)`	`<int>`	`<int>`$_1$ − `<int>`$_2$
`mul(<int>`$_1$`,<int>`$_2$`)`	`<int>`	`<int>`$_1$ × `<int>`$_2$
`div(<int>`$_1$`,<int>`$_2$`)`	`<int>`	`<int>`$_1$ / `<int>`$_2$
`mod(<int>`$_1$`,<int>`$_2$`)`	`<int>`	`<int>`$_1$ **mod** `<int>`$_2$
`abs(<int>`$_1$`)`	`<int>`	absolute value | `<int>`$_1$ |
`radd(<real>`$_1$`,<real>`$_2$`)`	`<real>`	`<real>`$_1$ + `<real>`$_2$
`rsub(<real>`$_1$`,<real>`$_2$`)`	`<real>`	`<real>`$_1$ − `<real>`$_2$
`rmul(<real>`$_1$`,<real>`$_2$`)`	`<real>`	`<real>`$_1$ × `<real>`$_2$
`rdiv(<real>`$_1$`,<real>`$_2$`)`	`<real>`	`<real>`$_1$ / `<real>`$_2$
`rabs(<real>`$_1$`)`	`<real>`	absolute value |`<real>`$_1$|
`sin(<real>`$_1$`)`	`<real>`	sin(`<real>`$_1$)
`cos(<real>`$_1$`)`	`<real>`	cos(`<real>`$_1$)
`atan(<real>`$_1$`)`	`<real>`	tan^{-1}(`<real>`$_1$) in range $[-\pi/2, \pi/2]$
`atan2(<real>`$_1$`, <real>`$_2$`)`	`<real>`	tan^{-1}(`<real>`$_1$/`<real>`$_2$) in range $[-\pi, \pi]$
`exp(<real>`$_1$`)`	`<real>`	exponential function $e^{\texttt{<real>}_1}$
`log(<real>`$_1$`)`	`<real>`	natural logarithm $ln(\texttt{<real>}_1)$, $\texttt{<real>}_1 > 0$
`log10(<real>`$_1$`)`	`<real>`	base 10 logarithm $log_{10}(\texttt{<real>}_1)$, $\texttt{<real>}_1 > 0$
`sqrt(<real>`$_1$`)`	`<real>`	$\sqrt{\texttt{<real>}_1}$, $\texttt{<real>}_1 \geq 0$

C.3 Functions on lists and multi-sets

Function	Result type	Description
`first(<list>`$_1$`)`	`<term>`	first element of `<list>`$_1$; `[]` for non-lists
`next(<list>`$_1$`)`	`<list>`	remaining elements of `<list>`$_1$; `[]` for non-lists
`join(<term>`$_1$`,<term>`$_2$`)`	`<list>`	concatenation of `<term>`$_1$ and `<term>`$_2$; for a list argument `<term>`$_i$ ($i = 1, 2$), the list elements are spliced into the new list; non-list arguments are included as single element of the new list.
`size(<list>`$_1$`)`	`<int>`	|`<list>`$_1$| (number of elements in list)
`index(<list>`$_1$`,<int>`$_1$`)`	`<term>`	If |`<list>`$_1$| $\leq$ `<int>`$_1$ return the `<int>`$_1$th element from `<list>`$_1$; otherwise `[]` and give a warning.
`replace(<list>`$_1$`,<int`$_1$`>,<term>`$_1$`)`	`<list>`	If |`<list>`$_1$| $\leq$ `<int>`$_1$ replace the `<int>`$_1$th element of `<list>`$_1$ by `<term>`$_1$ and return the modified (and partially copied) version of `<list>`$_1$; otherwise return `<list>`$_1$ and give a warning.
`get(<list>`$_1$`,<term>`$_1$`)`	`<term>`	If `<list>`$_1$ contains a pair `[<term>`$_1$`, <term>`$'_1$`]` then `<term>`$'_1$; otherwise `[]`.
`put(<list>`$_1$`,<term>`$_1$`, <term>`$_2$`)`	`<list>`	If `<list>`$_1$ contains a pair `[<term>`$_1$`, <term>`$'_1$`]` then replace it by `[<term>`$_1$`, <term>`$_2$`]`; otherwise add a new pair `[<term>`$_1$`, <term>`$_2$`]` to `<list>`$_1$.
`member(<term>`$_1$`,<list>`$_2$`)`	`<bool>`	`<term>`$_1$ $\in$ `<list>`$_2$ (membership in multi-set)
`subset(<list>`$_1$`, <list>`$_2$`)`	`<bool>`	`<list>`$_1$ $\subseteq$ `<list>`$_2$ (subset on multi-sets)
`diff(<list>`$_1$`, <list>`$_2$`)`	`<list>`	`<list>`$_1$ $-$ `<list>`$_2$ (difference on multi-sets)
`inter(<list>`$_1$`, <list>`$_2$`)`	`<list>`	`<list>`$_1$ $\cap$ `<list>`$_2$ (intersection on multi-sets)

C.4 Predicates and functions on terms

Function	Result type	Description
`is-bool(<term>)`	`<bool>`	If `<term>` is of type `bool` then `true`; otherwise `false`.
`is-int(<term>)`	`<bool>`	If `<term>` is of type `int` then `true`; otherwise `false`.
`is-real(<term>)`	`<bool>`	If `<term>` is of type `real` then `true`; otherwise `false`.
`is-str(<term>)`	`<bool>`	If `<term>` is of type `str` then `true`; otherwise `false`.
`is-bstr(<term>)`	`<bool>`	If `<term>` is of type `bstr` then `true`; otherwise `false`.
`is-appl(<term>)`	`<bool>`	If `<term>` is an application then `true`; otherwise `false`.
`is-list(<term>)`	`<bool>`	If `<term>` is a list then `true`; otherwise `false`.
`is-empty(<term>)`	`<bool>`	If `<term>` equals `[]` then `true`; otherwise `false`.
`is-var(<term>)`	`<bool>`	If `<term>` is a variable then `true`; otherwise `false`.
`is-result-var(<term>)`	`<bool>`	If `<term>` is a result variable then `true`; otherwise `false`.
`is-formal(<term>)`	`<bool>`	If `<term>` is a formal variable then `true`; otherwise `false`.
`fun(<term>)`	`<str>`	If `<term>` is an application then its function symbol; otherwise `""`.
`args(<term>)`	`<list>`	If `<term>` is an application then its argument list; otherwise `[]`.

C.5 Miscellaneous functions

Function	Result type	Description
`process-id`	`<int>`	id of current process
`process-name`	`<str>`	name of current process
`quote(<term>)`	`<term>`	quoted (unevaluated) term, only variables are replaced by their value
`functions`	`<list>`	list of built-in functions
`current-time`	`<list>`	six-tuple describing current absolute time
`sec(<int>`$_1$`)`	`<int>`	convert `<int>`$_1$ in seconds
`msec(<int>`$_1$`)`†	`<int>`	convert `<int>`$_1$ in milli-seconds

†Not yet implemented in the current version

D
Concrete Syntax of ATerms

A formal definition of the concrete syntax of ATerms using the syntax definition formalism SDF [HHKR92] is presented here. Note that there is no concrete syntax defined for blobs, because a humanly readable representation of blobs depends on the type of data stored in the blob.

```
hiddens
  sorts EscChar AFunChar ATerms Annotation
  lexical syntax
    "\\" ~[]                          -> EscChar
    "\\" [01][0-7][0-7]               -> EscChar
    ~[\000-\040"\\]                   -> AFunChar
    EscChar                           -> AFunChar

  context-free syntax
    ATerm                             -> ATerms
    ATerm "," ATerms                  -> ATerms
    "{" ATerms "}"                    -> Annotation

exports
  sorts ATerm ATermList ATermAppl ATermInt
        ATermReal ATermPlaceholder AFun

  lexical syntax
    [ \n\t]                           -> LAYOUT

    [a-zA-Z][a-zA-Z0-9_\-\*\+]*       -> AFun
    "\"" AFunChar* "\""               -> AFun

    [0-9]+                            -> ATermInt
    "-" ATermInt                      -> ATermInt
   ATermInt "." [0-9]+                -> ATermReal
```

```
    ATermInt "." [0-9]+ "e" ATermInt     -> ATermReal

  context-free syntax
    AFun                                 -> ATermAppl
    AFun "(" ATerms ")"                  -> ATermAppl

    "[" "]"                              -> ATermList
    "[" ATerms "]"                       -> ATermList

    "<" ATerm ">"                        -> ATermPlaceholder

    ATermAppl                            -> ATerm
    ATermAppl Annotation                 -> ATerm
    ATermList                            -> ATerm
    ATermList Annotation                 -> ATerm
    ATermInt                             -> ATerm
    ATermInt Annotation                  -> ATerm
    ATermReal                            -> ATerm
    ATermReal Annotation                 -> ATerm
    ATermPlaceholder                     -> ATerm
    ATermPlaceholder Annotation          -> ATerm
```

E

Level 2 Interface for ATerms

The operations described in Section 6.2 are not sufficient for all applications. Some applications need more control over the underlying implementation, or need operations that can be implemented using level one constructs but can be expressed more concisely and implemented more efficiently using more specialized constructs.
We have therefore designed a level 2 interface that is a strict superset of the level 1 interface described in Section 6.2. Some new datatypes are introduced, as well as some new operations on ATerms.

The level 2 interface introduces 7 new datatypes. Except for the auxiliary datatype AFun for representing function symbols, they are subtypes of the ATerm datatype, and implement the different term types. These subtypes allow us to introduce operations that are only valid for one specific term type, instead of the general ATerm operations described earlier.

ATermInt: This datatype represents integer terms. The operations on ATermInt are:

- `ATermInt ATmakeInt(Integer` v`)`: Construct a new integer term corresponding to the integer value v.
- `Integer ATgetInt(ATermInt` i`)`: Retrieve the value of an integer term.

ATermReal: This datatype represents real-number terms. The operations on ATermReal are:

- `ATermReal ATmakeReal(Real` v`)`: Construct a new real term.
- `Real ATgetReal(ATermReal` r`)`: Retrieve the value of a real term.

AFun: An AFun consists of a string defining the function name, an arity, and an indication whether the symbol name is quoted or not. The operations on symbols are:

- `AFun ATmakeAFun(String` nm`, Integer` ar`, Boolean` q`)`: Construct a new symbol. If a symbol with the given name nm, arity ar, and quotation q already exists, the existing symbol is returned. Otherwise a new symbol is created and returned. AFuns are also subject to garbage collection in order to avoid long running (interactive) programs from slowly running out of symbols.

- `String ATgetName(AFun` s`)`: Retrieve the name of symbol s.
- `Integer ATgetArity(AFun` s`)`: Retrieve the arity of a symbol.
- `Boolean ATisQuoted(AFun` s`)`: Check if a symbol is quoted.

ATermAppl: This datatype represents function applications consisting of a function symbol and a number of arguments. The operations on this datatype are:

- `ATermAppl ATmakeAppl`n`(AFun` f`, ATerm` a_0`, ..., ATerm` a_{n-1}`)`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to construct a new function application with the given function symbol f and arguments.
- `ATermAppl ATmakeAppl(AFun` f`, ATermList` as`)`: Construct a new function application with the given function symbol f and a list of arguments $args$
- `AFun ATgetFun(ATermAppl` ap`)`: Retrieve the function symbol of a function application.
- `ATerm ATgetArgument(ATermAppl` ap`, Integer` n`)`: Retrieve a specific argument.

ATermList: This datatype represents the binary list constructor. Element indices start at 0. Thus a list of length n has elements $0, \ldots, n-1$. The operations on ATermList are:

- `ATermList ATmakeList`n`(ATerm` e_0`,..., ATerm` e_{n-1}`)`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to quickly construct small lists of terms.
- `Integer ATgetLength(ATermList` l`)`: Retrieve the length of l.
- `ATerm ATgetFirst(ATermList` l`)`: Retrieve the first element of list l.
- `ATermList ATgetNext(ATermList` l`)`: Retrieve all but the first element of list l.
- `ATermList ATgetPrefix(ATermList` l`)`: Retrieve all but the last element of list l.
- `ATerm ATgetLast(ATermList` l`)`: Retrieve the last element from list l.
- `ATermList ATgetSlice(ATermList` l`, Integer` $from$`, Integer` to`)`: Retrieve the portion of list l from position $from$ through $to - 1$.
- `Boolean ATisEmpty(ATermList` l`)`: Check if list l contains zero elements.

- `ATermList ATinsert(ATermList` l`, ATerm` e`)`: Insert a single element e at the start of list l.
- `ATermList ATinsertAt(ATermList` l`, ATerm` e`, Integer` i`)`: Insert a single element e at position i in list l.
- `ATermList ATappend(ATermList` l`, ATerm` e`)`: Append a single element e to the end of list l.
- `ATermList ATconcat(ATermList` l_1`, ATermList` l_2`)`: Concatenate lists l_1 and l_2.
- `Integer ATindexOf(ATermList` l`, ATerm` e`, Integer` i`)`: Search for an element e in list l and return the index of the first location where e is present. Start searching at index i. If the element is not present, return -1.
- `Integer ATlastIndexOf(ATermList` l`, ATerm` e`, Integer` i`)`: Search backwards for element e in list l, and return the index of the last location where the element is present. Start searching at index i. If the element is not present, return -1.
- `ATerm ATelementAt(ATermList` l`, Integer` i`)`: Retrieve element at position i from list l.
- `ATermList ATremoveElement(ATermList` l`, ATerm` e`)`: Remove once occurrence of element e from list l.
- `ATermList ATremoveElementAt(ATermList` l`, Integer` i`)`: Remove the element at position i from list l.

ATermPlaceholder: This datatype represents placeholder terms. The operations on this datatype are:

- `ATermPlaceholder ATmakePlaceholder(ATerm` tp`)`: Construct a new placeholder term.
- `ATerm ATgetPlaceholder(ATermPlaceholder` ph`)`: Retrieve the type of this placeholder.

ATermBlob: This datatype represents Binary Large OBject terms. The operations on ATermBlob are:

- `ATermBlob ATmakeBlob(Integer` n`, Data` d`)`: Construct a new blob term of size n and containing data d.
- `Integer ATgetBlobSize(ATermBlob` b`)`: Retrieve the size of blob b.

- `Data ATgetBlobData(ATermBlob` *`blob`*`)`: Retrieve the data pointer stored in blob *b*.

The memory management of blobs must be done explicitly by the application programmer.

Auxiliary: The level two interface provides functionality to create and manipulate user-defined hash tables.

Bibliography

[ADR95] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[AG93] A.W. Appel and M.J.R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[Alb97] W. van Albada. Debugging compiled Asf+Sdf specifications using hybrid functions. Master's thesis, Programming Research Group, University of Amsterdam, 1997.

[All78] J.R. Allen. *Anatomy of LISP*. McGraw-Hill, 1978.

[AM87] A.W. Appel and D. MacQueen. A standard ML compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, LNCS, pages 301–324, 1987.

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models*, LNCS, pages 34–56. Springer-Verlag, 1996.

[ASN95] Information Technology – Abstract Syntax Notation One (ASN.1): Encoding Rules – Packed Encoding Rules (PER). Technical report, International Telecommunication Union, 1995. ITU-T Recommendation X.691.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Aus90] D. Austry. The VTP project: modular abstract syntax specification. Rapport de Recherche 1219, INRIA, Sophia-Antipolis, 1990.

[AY91] D.K. Arvind and D. Yokotsuka. Debugging concurrent programs using static analysis and run-time hardware monitoring. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 716–719. IEEE Computer Society Press, 1991.

[BCD$^+$89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *SIGPLAN Notices*, 24(2):14–24, 1989. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. The Centaur Tutorial is available at http://www.inria.fr/croap/centaur/tutorial/main/main.html.

[BDK$^+$96] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and A.E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Bea83] B. Beander. An interactive, symbolic, multilingual debugger. In *Symposium on High-Level Debugging, 1983*, pages 173–180. ACM SIGSOFT/SIGPLAN, 1983.

[Ber91] D. Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edingburgh, 1991.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[BHK97] M.G.J. van den Brand, J. Heering, and P. Klint. Renovation of the ASF+SDF Meta-Environment—current state of affairs. Technical report, University of Amsterdam, Programming Research Group, Amsterdam, November 1997.

[BHKO00] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The Asf+Sdf compiler. Technical Report SEN-R0014, Centrum voor Wiskunde en Informatica (CWI), 2000. Submitted.

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.

[BK98] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BKK$^+$96] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1996.

[BKMO97] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new Asf+Sdf Meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, November 1997. Springer-Verlag.

[BKO98] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Aterms: Exchanging data between heterogeneous tools for CASL. Note T-3, in [CoF98], 1998.

[BKO99] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 198–213, 1999.

[BKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[BKV98] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 139–161. Elsevier Science, 1998.

[BMS87] R. Bahlke, B. Moritz, and G. Snelting. A generator for language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'97 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *SIGPLAN Notices*, pages 92–101, 1987.

[Boe93a] H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the conference on Programming language design and implementation*, pages 197–206. ACM Press, 1993.

[Boe93b] H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 28, 6, pages 197–206, June 1993.

[Bru96] J.J. Brunekreef. A transformation tool for pure Prolog programs. In J.P. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, volume 1207 of *LNCS*, pages 130–145. Springer-Verlag, 1996.

[BSV97] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[BV96] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[BW83] P.C. Bates and J.C. Wileden. High level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software, 3(4)*, pages 394–399, 1983.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience (SPE)*, 18(9):807–820, 1988.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[CBM90] W.H. Cheung, J.P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, pages 106–115, 1990.

[Cha96] D. Chappell. *Understanding ActiveX(TM) and OLE*. MicroSoft Press, 1996.

[CL98] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [CoF98], 1998.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, third, revised and extended edition, 1987.

[CoF98] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[1] and FTP[2], 1998.

[Cor99] *The Common Object Request Broker: Architecture and Specification*. Object Managament Group (OMG), 1999.

[Deu94] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.

[DFG$^+$94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *International Conference on Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, 1994.

[DG95] D. Dams and J.F. Groote. Specification and implementation of components of a μCRL toolbox. Technical Report 152, Utrecht University, 1995.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[1]http://www.brics.dk/Projects/CoFI
[2]ftp://ftp.brics.dk/Projects/CoFI

[Dik89] C.H.S. Dik. A fast implementation of the Algebraic Specification Formalism. Master's thesis, University of Amsterdam, Programming Research Group, 1989.

[dJ99] H.A. de Jong. A Visualization Framework for ToolBus Applications. Master's thesis, Programming Research Group, University of Amsterdam, 1999.

[DK98] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[DKT96] A. van Deursen, P. Klint, and F. Tip. Origin tracking and its applications. In Deursen et al. [DHK96], pages 249–294.

[EKN96] J. Ellson, E. Koutsofios, and S. C. North. *graphviz* package, release 1.2, 1996. Software package available from http://www.research.att.com/sw/tools/graphviz/.

[EN96] J. Ellson and S. C. North. TclDG - a Tcl extension for dynamic graphs. In *Proceedings of the fourth annual Tcl/Tk workshop*. USENIX Association, 1996.

[FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., 1995.

[FKW98] W. Fokkink, J.F.Th. Kamperman, and H.R. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, 1998.

[For89] A. Forin. Debugging of heterogeneous parallel systems. In *Workshop on Parallel and Distributed Debugging, 1988*, pages 130–140. ACM, 1989.

[GB94] C. Groza and M.G.J. van den Brand. The algebraic specification of annotated abstract syntax trees. Technical Report P9414, University of Amsterdam, Programming Research Group, 1994.

[Gel85] D. Gelernter. Generative communication in linda. In *ACM Transactions on Programming Languages and Systems*, volume 7(1), pages 80–112, 1985.

[GHL$^+$92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

[GL99] J.F. Groote and B. Lisser. Tutorial and reference guide for the μCRL toolset version 1.0. Technical report, CWI, Amsterdam, 1999.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.

[Gro92] J. Grosch. Ast – a generator for abstract syntax trees. Technical Report 15, GMD Karlsruhe, 1992.

[H$^+$96] P.H. Hartel et al. Benchmarking implementations of functional languages with 'pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6:621–655, 1996.

[Ham99] G. Hamilton, editor. *JavaBeans*. Sun Microsystems, 1999.

[Han99a] D.R. Hanson. Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.

[Han99b] D.R. Hanson. A machine-independent debugger - revisited. *Software - Practice and Experience*, 29(10):849–862, 1999.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[HK95] J. Heering and P. Klint. The prehistory of ASF+SDF (1980–1984). In M.G.J. van den Brand, A. van Deursen, T. B. Dinesh, J. F. Th. Kamperman, and E. Visser, editors, *ASF+SDF'95: A Workshop on Generating Tools from Algebraic Specifications*, Technical Report P9504, pages 1–4. Programming Research Group, University of Amsterdam, 1995.

[HK97] D.R. Hanson and J.L. Korn. A simple and extensible graphical debugger. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–184, 1997.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[HR96] D.R. Hanson and M. Raghavachari. A machine-independent debugger. *Software - Practice and Experience*, 26(11):1277–1299, 1996.

[JHH+93] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler. The Glasgow Haskell compiler: a technical overview. *Proc. Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, 1993.

[JL96] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[Kam94] J.F.Th. Kamperman. GEL, a graph exchange language. Technical Report CS-R9440, CWI, Amsterdam, 1994.

[Kap87] S. Kaplan. A compiler for conditional term rewriting systems. In P. Lescanne, editor, *Proceedings of the First International Conference on Rewriting Techniques*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 1987.

[Kar98] M. Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of the ACM SIGSOFT sixth International Symposium on Foundations of Software Engineering*, pages 131–142, 1998.

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Report P9312, Programming Research Group, University of Amsterdam, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KN93] E. Koutsofios and S. C. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ., 1993.

[Knu73] D. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[Koo94] J.W.C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments*. PhD thesis, University of Amsterdam, 1994.

[KR88] B.W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall, second edition, 1988.

[Kui96] T. Kuipers. Language independent structure editing using the ToolBus. Master's thesis, Programming Research Group, University of Amsterdam, 1996. Available from ftp://ftp.wins.uva.nl/pub/programming-research/MasterTheses/Kuipers.ps.gz.

[Kun95] T. Kunz. High-level views of distributed executions. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, 1995.

[KV94] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proceedings ASMICS Workshop on Parsing Theory*, pages 1–20, 1994. Published as Technical Report 126–1994, Computer Science Department, University of Milan.

[Lam87] D.A. Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, 1987.

[LeL90] INRIA, Rocquencourt. *LeLisp, Version 15.23, Reference Manual*, 1990.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, unix programmer's supplementary documents, volume 1 (ps1) edition, 1986.

[Lut99] S.P. Luttik. Description and formal specification of the link layer protocol (SEN-R9706). Technical report, CWI, Amsterdam, 1999.

[Meu94] E.A. van der Meulen. *Incremental Rewriting*. PhD thesis, University of Amsterdam, 1994.

[NK94] S. C. North and E. Koutsofios. Applications of graph visualization. *Graphics Interface '94*, pages 235–245, 1994.

[Nor96] S. C. North. Incremental layout in dynaDAG. In *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996.

[Oli96a] P. Olivier. Embedded system simulation: testdriving the toolbus. Technical Report P9601, University of Amsterdam, Programming Research Group, 1996.

[Oli96b] P.A. Olivier. A simulator framework for embedded systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION)*, volume 1061 of *LNCS*, pages 436–439. Springer-Verlag, 1996.

[Oli97] P.A. Olivier. Debugging distributed applications using a coordination architecture. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (COORDINATION)*, volume 1061 of *LNCS*, pages 98–114. Springer-Verlag, 1997.

[OMG97] OMG. The common object request broker: Architecture and specification, revision 2,0. Technical Report 97-02-25, Object Management Group, 1997. Available at: `http://www.omg.org`.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[PE94] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Concurrent Clean - version 1.0 - Language Reference Manual, draft version.* Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands, 1994.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[RH92] N. Ramsey and D.R. Hanson. A retargetable debugger. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN Notices*, pages 22–31, 1992.

[RvRT89] R. van Staveren R. van Renesse and A.S. Tanenbaum. Performance of the amoeba distributed operating system. *Software – Practice and Experience 19*, 1989.

[Sos95] Rok Sosič. A procedural interface for program directing. *Software - Practice and Experience*, 25(7):767–787, July 1995.

[SP91] R.M. Stallman and R.H. Pesch. *Using GDB: A guide to the GNU Source-Level Debugger.* Free Software Foundation/Cygnus Support, July 1991.

[SPSS00] R. Stallman, R. Pesch, and et al. S. Shebs. *Debugging with GDB*, 2000.

[Szy97] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[Tip91] F. Tip. The equation debugger. Master's thesis, University of Amsterdam, Programming Research Group, 1991.

[Tip95] F. Tip. *Generation of Program Analysis Tools.* PhD thesis, University of Amsterdam, Programming Research Group, 1995. prof. dr. P. Klint.

[TK90] M. Terashima and Y. Kanada. HLisp—its concept, implementation and applications. *Journal of Information Processing*, 13(3):265–275, 1990.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[VBT98] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

[Vis95] E. Visser. *ASF+SDF to LaTeX: The User Manual.* Programming Research Group, University of Amsterdam, 1995. http://adam.fwi.uva.nl/~visser/tolatex/.

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 22 June 1990.

[WAKS97] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[WFW$^+$94] R.P. Wilson, R.S. French, Ch.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K.Tjiang, Shih-Wei Liao, Chau-Wen Tseng, M.W. Hall, M.S. Lamm, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizeing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[WvRA96] A. Watters, G. van Rossum, and J. Ahlstrom. *Internet Programming with Python*. MIS Press/Henry Holt publishers, 1996.

[XML98] Extensible markup language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. Available at: `http:www.w3.org/TR/REC-xml`.

Samenvatting

Een krachtig middel om de complexiteit van software-systemen te verminderen is het gebruik van domeinspecifieke talen. Dit soort talen biedt constructies die dichter bij het probleemdomein liggen dan het geval is bij meer 'algemene' programmeertalen. Een van de grootste problemen met domeinspecifieke talen is het gebrek aan goede ontwikkelomgevingen. Traditionele talen hebben meestal een breed draagvlak, en bestaan vaak al tientallen jaren. Hierdoor worden dit soort talen meestal ondersteund door een breed scala aan geavanceerde ontwikkelgereedschappen die de programmeur helpen bij het bouwen van grote programma's. De ondersteuning voor domeinspecifieke talen is vaak van veel mindere kwaliteit. Doordat het draagvlak kleiner is, bijvoorbeeld beperkt tot één organisatie, is er gewoon geen tijd en geld om op een traditionele manier geavanceerde hulpmiddelen te ontwikkelen.

In principe zijn er twee manieren om de ontwikkelkosten van dit soort hulpmiddelen laag te houden: door gebruik te maken van generieke hulpmiddelen die geparametriseerd kunnen worden, of door specialistische hulpmiddelen direct te genereren uit een formele taaldefinitie. De eerste aanpak is het meest bekend in relatief eenvoudig gereedschap zoals syntax gestuurde editors en pretty printers. In het eerste deel van dit proefschrift wordt aangetoond dat deze aanpak voor een complex stuk software als een interactieve debugger[3] ook prima mogelijk is om een generieke variant te ontwikkelen die geïnstantieerd kan worden voor een specifieke programmeertaal.

Het direct genereren van deze hulpmiddelen uit een formele (programmeer)taaldefinitie is een minder bekende aanpak. Deze techniek ligt aan de basis van de ASF+SDF Meta-omgeving. Deze geïntegreerde ontwikkelomgeving richt zich op het automatisch genereren van software hulpmiddelen gebaseerd op formele definities van programmeertalen in het algebraïsche specificatieformalisme ASF+SDF. Deze Meta-omgeving is een hulpmiddel om programmeertalen en programmeergereedschap te ontwikkelen.

In het tweede deel van dit proefschrift komen een aantal implementatietechnieken aan de orde die gebruikt worden in de ASF+SDF Meta-omgeving. Bovendien wordt er een koppeling gemaakt met het eerste deel van dit proefschrift door te laten zien hoe de technieken voor generieke debugging uit het eerste deel gebruikt kunnen worden om een debugger te ontwikkelen voor ASF+SDF specificaties, en om ondersteuning voor debuggen te bieden voor talen

[3] Een *debugger* (letterlijk: *ontluizer*) is een programma dat helpt bij het detecteren en repareren van fouten in software.

gespecificeerd met behulp van ASF+SDF.

Generiek debuggen

Implementaties van (moderne) programmeertalen bieden bijna altijd ondersteuning voor het debuggen op broncode niveau. Om de implementatie flexibel en eenvoudig te houden betreft het hier meestal ondersteuning voor debuggen op een vrij laag niveau. Dit maakt het mogelijk om de echte debugger onafhankelijk te ontwikkelen van de taalimplementatie zelf. In dit proefschrift stel ik voor om bovenop deze laag-niveau ondersteuning een abstractielaag aan te brengen die abstraheert van de semantiek van de betreffende programmeertaal. Bovenop deze abstractielaag kunnen nu generieke hulpmiddelen voor debugging ontwikkeld worden die niet meer specifiek zijn voor de programmeertaal in kwestie. Dit betekent dat alleen deze abstractielaag voor iedere programmeertaal opnieuw ontwikkeld hoeft te worden. De eigenlijke hulpmiddelen voor debugging die normaal gesproken het meeste werk kosten kunnen keer op keer hergebruikt worden.

De ASF+SDF Meta-omgeving

Al het werk beschreven in dit proefschrift is gedaan in de context van de herimplementatie van de ASF+SDF Meta-omgeving. Deze herimplemenatie was nodig omdat de originele implementatie inmiddels hopeloos verouderd is. Modernisatie van deze oude implementatie was technisch niet meer haalbaar, o.a. door de monolithische opzet van het systeem. In de nieuwe implementatie is dan ook gekozen voor een ontwerp dat gebaseerd is op een componentsgewijze opsplitsing van de functionaliteit. Door tijdens het ontwerp veel aandacht aan de flexibiliteit van de individuele componenten te besteden, wordt het mogelijk om veel van deze componenten ook buiten de ASF+SDF Meta-omgeving in te zetten.

De ATerm bibliotheek

Alle componenten van de ASF+SDF Meta-omgeving maken voor hun implementatie gebruik van de ATerm bibliotheek, en de beschrijving van de principes achter deze bibliotheek neemt dan ook een belangrijke plaats in in dit proefschrift.

Veel gegevens kunnen het meest natuurlijk worden gerepresenteerd als boomstructuren. Dit geldt zeker voor de gegevens die gebruikt worden bij de implementatie van programmeeromgevingen. Gegevensstructuren als parseerbomen, abstracte syntaxbomen en parseertabellen kunnen uitstekend als boomstructuren gerepresenteerd worden. Een andere eigenschap van deze gegevens is dat ze vaak veel redundante informatie bevatten. Als we bijvoorbeeld kijken naar COBOL programma's, zien we zeer veel codeduplicatie optreden. Deze duplicatie is in de parseerbomen en in de abstracte syntaxbomen terug te vinden als het bestaan van identieke takken.

We hebben de ATerm bibliotheek ontwikkeld als een efficiënte implementatie van dit soort boomstructuren. Om deze bibliotheek zowel qua geheugen als qua tijd efficiënt te krijgen, maken we gebruik van een techniek genaamd 'maximale sharing'. Deze techniek houdt in dat we nooit twee keer dezelfde boom opbouwen. Bij het creëren van een nieuwe boom kijken we eerst of we die boom al in het geheugen hebben. Zo ja, dan wordt de bestaande boom teruggegeven, anders wordt er een nieuwe boom aangemaakt.

Deze techniek brengt natuurlijk extra werk met zich mee, omdat bij iedere boomcreatie gekeken moet worden of de betreffende boom al bestaat. Bovendien is het niet meer mogelijk termen te wijzigen nadat ze zijn gecreëerd. Dit betekent dat alle operaties een functioneel gedrag vertonen. Gelukkig krijgen we ook iets terug voor deze extra inspanningen, omdat het vergelijken of twee bomen identiek zijn opeens heel goedkoop wordt: twee bomen zijn immers alleen identiek als ze hetzelfde object zijn, dus met hetzelfde adres in het geheugen. Bovendien bespaart deze techniek enorm op het aantal object creaties, waardoor ook bespaard wordt op de tijd die nodig is om niet gebruikte objecten op te ruimen.

De ASF+SDF naar C compiler

ASF+SDF is een *executeerbaar* formalisme, waarbij de semantiek van dit formalisme gebaseerd is op termherschrijven. Aangezien veel van de componenten van de nieuwe ASF+SDF Meta-omgeving zelf geschreven zijn in ASF+SDF is de efficiënte executie van ASF+SDF specificaties van zeer groot belang voor dit project. Om ASF+SDF specificaties op een efficiënte manier uit te kunnen voeren hebben we een compiler ontwikkeld die ASF+SDF vertaalt naar de programmeertaal C. Opvallend is dat we er met succes voor gekozen hebben om deze compiler zelf in ASF+SDF te specificeren.

Aangezien de ASF+SDF Meta-omgeving bedoeld is om grote industriële projecten aan te kunnen, is het ook voor de compiler belangrijk dat die met grote specificaties om kan gaan. Bovendien moeten gecompileerde specificaties overweg kunnen met grote hoeveelheden data. Om dit te bewerkstelligen maakt de gegenereerde C code gebruik van de hierboven beschreven ATerm bibliotheek. Hierdoor wordt alle redundantie direct uitgebuit. Bovendien is één van de meest voorkomende operaties in de gegenereerde code het vergelijken van termen: typisch iets waar de ATerm bibliotheek goed mee overweg kan.

Het debuggen van ASF+SDF specificaties

Specificeren is niets anders dan programmeren in een formele taal, dat wil zeggen een taal met een solide wiskundige onderbouwing. Dit wil zeggen dat het schrijven van goede specificaties net zo moeilijk (of soms zelfs moeilijker) is als het schrijven van goede programma's. Het is daarom dan ook belangrijk dat de specificatieschrijver de beschikking heeft over een goede debugger. Uiteraard is dit een prima kans om ons generieke debugsysteem nuttig in te zetten. We hebben hiertoe een koppeling gemaakt tussen ons generieke debugsysteem en de interpreter in de ASF+SDF Meta-omgeving, waardoor we met een beperkte inspanning direct een debugger voor ASF+SDF in handen hebben.

Als laatste presenteren we een zogenaamde "case-study" die laat zien hoe het mogelijk is om ook ondersteuning voor debugging te bieden voor programma's die uitgevoerd worden door een interpreter die gespecificeerd is in ASF+SDF. Deze case-study laat zien dat ASF+SDF samen met generiek debuggen een zeer krachtige combinatie vormt.

Titles in the IPA Dissertation Series