A Framework for Debugging

Olivier, P.A.

Citation for published version (APA):
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
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
Motivation / 1.1

Motivation

(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 icc [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 [Ali78] 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

- 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 [BK099] 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 implementation 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 Icc 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.
INTRODUCTION

The ASF+SDF Meta-Environment implementation (Chapter 5, [BKM097]) 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.