A Framework for Debugging
Olivier, P.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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 data structures 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 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
Debugging in the Context of ASF+SDF / 9.3

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.