DDD—A Free Graphical Front-End for UNIX Debuggers

Andreas Zeller and Dorothea Lütkehaus

Informatik-Bericht No. 95-07
7. August 1995

DDD—A Free Graphical Front-End for UNIX Debuggers

Andreas Zeller and Dorothea Lütkehaus*
Abteilung Softwaretechnologie
Technische Universität Braunschweig, Germany

Abstract

The Data Display Debugger (DDD) is a novel graphical user interface to GDB and DBX, the popular UNIX debuggers. Besides "usual" features such as viewing source texts and breakpoints, DDD provides a *graphical data display*, where data structures are displayed as graphs. A simple mouse click dereferences pointers or reveals structure contents. Complex data structures can be explored incrementally and interactively, using automatic layout if preferred. Each time the program stops, the data display reflects the current variable values. DDD has been designed to compete with well-known commercial debuggers; however, it is free software, protected by the GNU general public license. In this paper, we give a quick presentation of DDD and describe its architecture and basic functionality from a technical point of view.

Key words: Software Engineering, Debugging Aids, Diagnostics, User Interfaces

1 Introduction

A key part of the compile-edit-debug cycle is the debugging phase. A specialized debugging tool (a *debugger*) can significantly help to examine the dynamic behaviour of a program. Debuggers can be viewed as program interpreters; they provide support for inspecting the execution state in a symbolic way, and they allow for executing programs step-by-step or until a specific condition arises.

In the UNIX world, the GNU debugger (GDB) [9] is one of the most popular debuggers. The basic GDB provides a large range of functionality for all debugging purposes. Unfortunately, the GDB user interface is not suitable for casual and inexperienced users, since GDB recognizes more than one hundred of basic commands, each with its own set of options and gadgets. Such users may prefer a graphical user interface, emphasizing the most frequently used commands and providing direct manipulation facilities. Various graphical user interfaces for GDB are available today, notably XXGDB [2], TGDB [8] and GUI for GDB [3]. These GDB extensions provide separate windows for viewing the current source code location, manipulating breakpoints, and invoking frequently needed commands through push buttons. The original GDB command interface remains for experienced users and complex tasks.

In the last few years, graphical debugging interfaces showed up another advantage, namely *graphical data displays*. The *SoftBench* program debugger [4] and the *CodeCenter* environment [1] introduced facilities to display program data as graphs, allowing for simple exploration of complex data structures. Unfortunately, each of these debuggers comes with its own environment and, which is worse, with its own proprietary compiler.

Being a research institute, we rely on powerful and frequently updated software tools while suffering from a low budget. What we longed for was a comfortable debugging environment for little or no cost. In earlier projects, we had already developed a simple graph editor, a library to visualize program and data structures, and an adaptive UNIX interprocess communication library. By reusing these packages, we found it feasible to develop a comfortable graphical user interface for GDB in less than ten man-months; to make it publicly available required another two man-months. The resulting product, called DDD for *Data Display Debugger* [6, 7] is a free full-fledged debugging environment with an estimated 15,000 to 20,000 users three months after its publication.

2 A Quick Glance at DDD

Let us first take a look at DDD, as it presents itself to the user. In figure 1, we see three windows. The *command window* realizes the basic GDB command line interface, reflecting its editing and completion capabilities. The *source window*, below, shows the source code as well as a breakpoint (indicated by `#1#`) and the current execution posi-

* Authors' current address: Technische Universität Braunschweig, Abteilung Softwaretechnologie, Gaußstr. 17, D-38092 Braunschweig, Germany. E-mail: ddd@ips.cs.tu-bs.de.

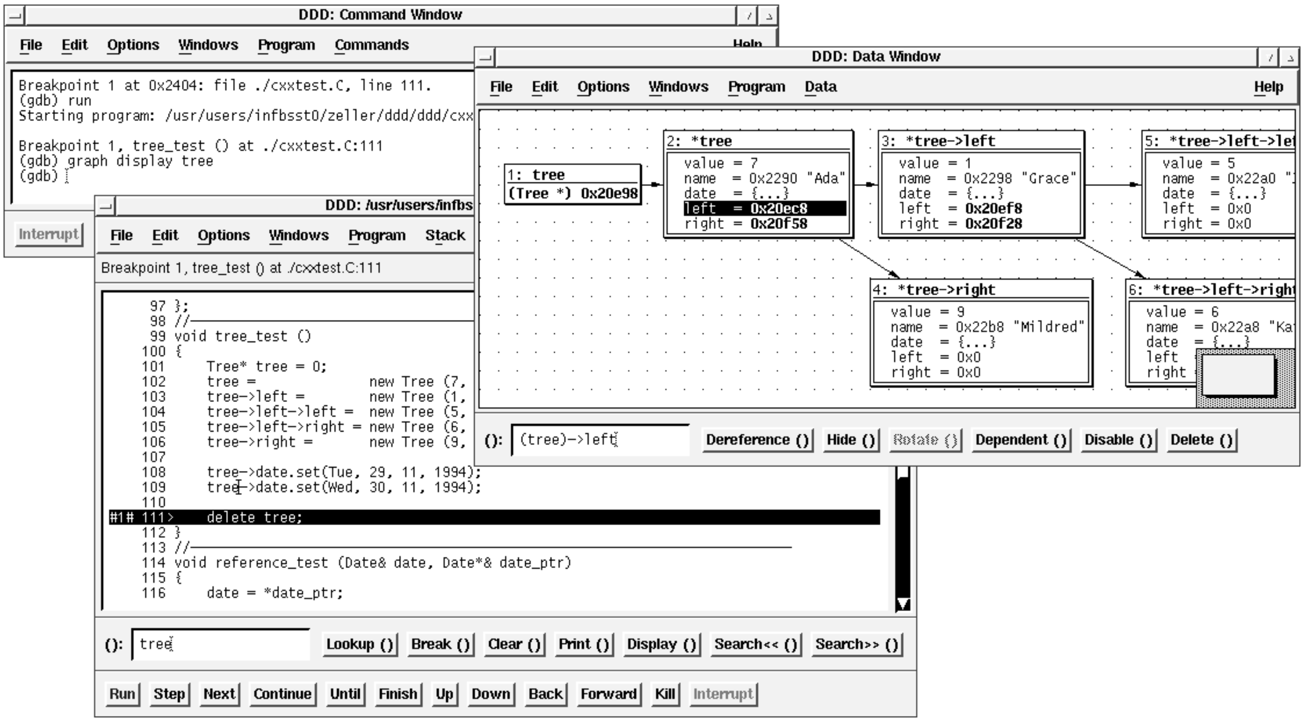

Figure 1: The Data Display Debugger

tion (highlighted). Arbitrary symbols and expressions can be selected in the program code and displayed by choosing a "Display" action from a popup menu or from the button box below.

Here is a simple example of DDD usage. User Lisa has executed a program up to a breakpoint and wishes to investigate the `date_ptrs` array, an array of pointers to `Date` record structures. She selects an occurrence of `date_ptrs` and chooses the `display` action, as shown in figure 2.

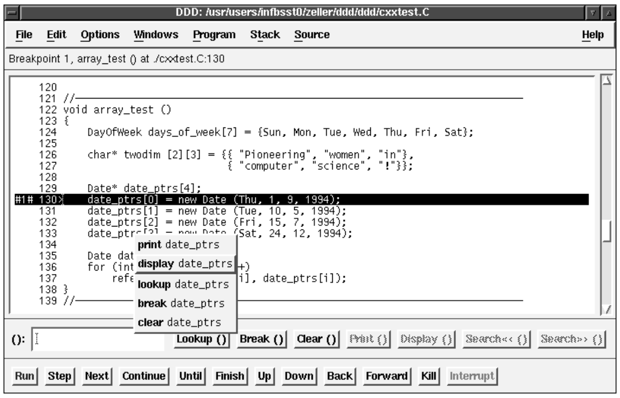

Figure 2: Selecting a symbol

The value of `date_ptrs` is displayed in a node in the data window. Since the size of `date_ptrs` is known at compile time and thus passed on to the debugger, GDB and DDD can show all four array elements (figure 3). If `date_ptrs` were a dynamically allocated array, then Lisa would have to specify the array size explicitly, using the GDB "artificial array" feature.

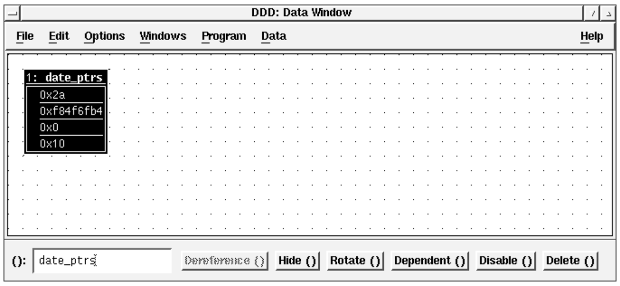

Figure 3: A data node

The array elements have not yet been initialized. This is done two lines later, which Lisa executes step-by-step by pressing the `Next` button. Each time the debugged program stops (or is stopped by GDB), the data display reflects the current variable values. Lisa focuses her interest on the second element, `date_ptrs[1]`, and selects `Dereference` from the popup menu (figure 4).

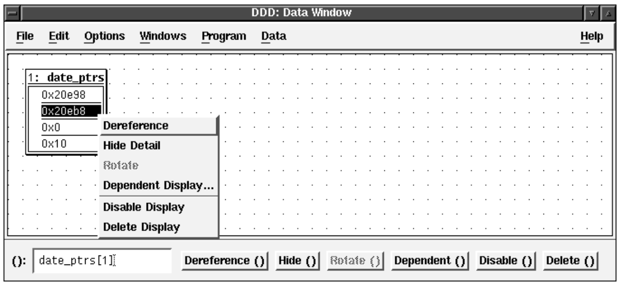

Figure 4: The node menu

The dereferenced element, `*date_ptrs[1]`, is shown in a *dependent node* (figure 5), a node referred from the originating node, `date_ptrs`.

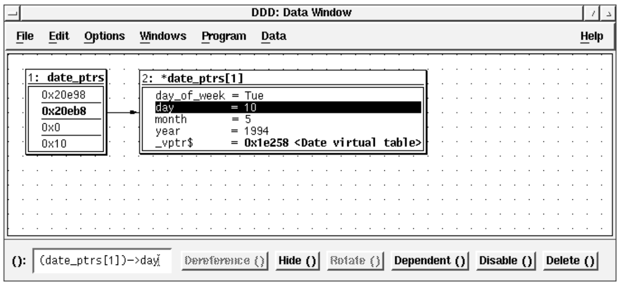

Figure 5: A dependent node

After the other elements have been initialized and displayed in a similar fashion, Lisa lets DDD layout the graph such that she can examine the entire `date_ptrs` structure, shown in figure 6.

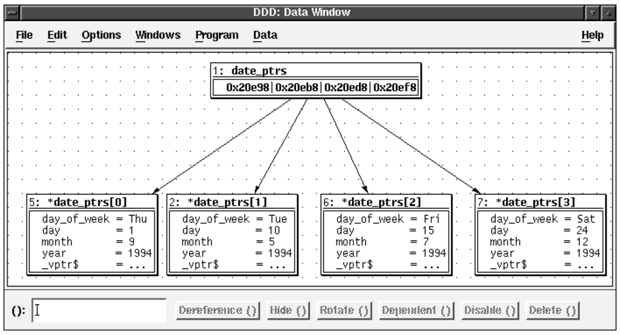

Figure 6: The entire `date_ptrs` structure

When the current function returns, `date_ptrs` no more exists; GDB and DDD automatically disable all nodes containing `date_ptrs`. They will be automatically re-enabled and shown with their current values the next time the current function is entered.

Besides the eye-pleasing data display, DDD provides today's standards of debugging environments as well as some unique features such as hypertext source navigation and lookup or debugging on remote hosts. In short, DDD has been designed to compete with well-known commercial debuggers.

3 A Debugger Front-End

When we started to design DDD, the first question that arised was: should we touch GDB code or not? The main problem was that free software like GDB undergoes far more changes than commercial software, simply because the source code is freely available. We thus decided to keep the coupling of DDD and GDB as low as possible, such that making changes to GDB does not necessarily imply recompiling or even changing DDD. Hence, DDD runs GDB as a separate process, interacting through the GDB command-line interface.

The communication channels between the user, DDD, GDB, and the debugged process are shown in figure 7. In order to minimize DDD response time, all communication between DDD and GDB is performed asynchronously. GDB commands issued from DDD are tagged with a callback routine and then placed into a command queue. The callback routine processes the GDB output when available.

For instance, if the user enters a GDB command manually, DDD tags the command with a callback routine that displays GDB output; as soon as the GDB command is

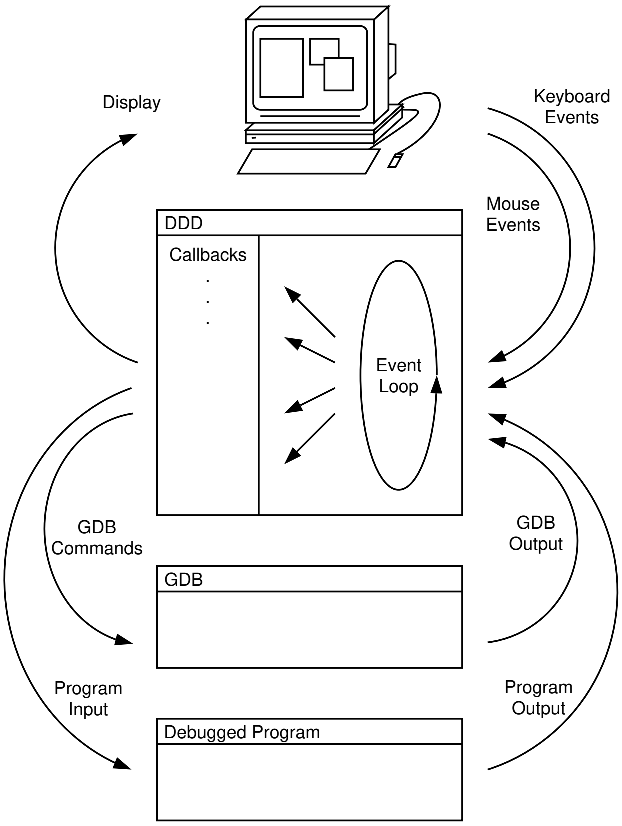

Figure 7: DDD Communication

completed, the callback routine is invoked and the GDB output is shown in the DDD command window.

The main DDD event loop simultaneously waits for user input, GDB output, and for GDB to become ready for input. When GDB becomes ready, the next command is taken from the queue and sent to GDB. Incoming GDB output is processed by the callback routine associated with the last command sent. This asynchronous scheme avoids DDD being blocked while waiting for GDB output; incoming events can be processed any time.

The separation of GDB and DDD processes makes DDD run slower, due to the time needed to interpret GDB responses. However, this approach turned out to have several advantages. For instance, we could replace GDB by debuggers whose source code was not available, such as Sun's DBX [10]. Another fortunate effect was that we could run DDD and GDB on separate machines, using a long-distance remote TTY connection, or run DDD and GDB in parallel on a multi-processor machine.

4 Boxes and Display Functions

All data displayed in the DDD data window is maintained by the underlying GDB debugger. GDB provides a *display list*, holding symbolic expressions to be evaluated and printed on standard output at each program stop. The GDB command `display tree` adds `tree` to the display list and makes GDB print the value of `tree` as `tree = (Tree *)0x20e98`, for instance, at each program stop. This GDB output is processed by DDD and displayed in the data window.

Each element of the display list, as transmitted by GDB, is read by DDD and translated into a *box*. Boxes are rectangular entities with a specific content that can be displayed in the data window. We distinguish *atomic* boxes and *composite* boxes. An atomic box holds white or black space, a line, or a string. Composite boxes are horizontal or vertical alignments of other boxes. Each box has a size and an extent that determines how it fits into a larger surrounding space.

Through construction of larger and larger boxes, DDD constructs a graph node from the GDB data structure in a similar way a typesetting system like TeX [5] builds words from letters and pages from paragraphs. In figure 8, we see how a framed text is built from five rectangular boxes. First, a horizontal alignment is built containing a vertical line, the text, and another vertical line. This composite box is then vertically aligned with two horizontal lines, resulting in a framed text.

Such constructions are easily expressed by means of functions mapping boxes onto boxes. These *display functions* can be specified by the user and interpreted by DDD, using an applicative language called VSL for *visual structure language* [11]. VSL functions can be specified by

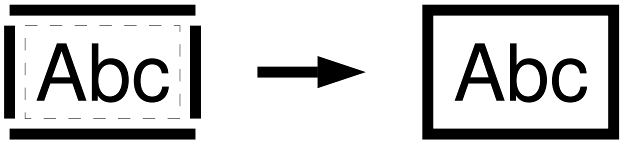

Figure 8: Building a frame from atomic boxes

the DDD user, leaving much room for extensions and customization. A VSL display function putting a frame around its argument looks like this:

```
// Put a frame around TEXT
frame(text) = hrule()
   | vrule() & text & vrule()
   | hrule();
```

Here, `hrule()` and `vrule()` are primitive functions returning horizontal and vertical lines, respectively. The `&` and | operators construct horizontal and vertical alignments from their arguments.

VSL provides basic facilities like pattern matching and variable numbers of function arguments. The `halign()` function, for instance, builds a horizontal alignment from an arbitrary number of arguments, matched by three dots (`...`):

```
// Horizontal alignment
halign(x) = x;
halign(x, ...) = x & halign(...);
```

Frequently needed functions like `halign()` are grouped into a standard VSL library.

5 Building Boxes from Data

To visualize data structures, each atomic type and each type constructor from the programming language is assigned a VSL display function. Atomic values like numbers, characters, enumerations, or character strings are displayed using string boxes holding their value; the VSL function to display them leaves them unchanged:

```
// Atomic Values
simple_value(value) = value;
```

Composite values require more attention. An array, for instance, may be displayed using a horizontal alignment:

```
// Array
array(...) = frame(halign(...));
```

When GDB sends DDD the value of an array, the VSL function `array()` is invoked with array elements as values. A GDB array expression `{1, 2, 3}` is thus evaluated in VSL as `array(simple_value("1"), simple_value("2"), simple_value("3"))`, which equals `"1" & "2" & "3"`, a composite box holding a horizontal alignment of three string boxes.

The actual VSL function used in DDD also puts delimiters between the elements and comes in a vertical variant as well, as shown in figure 9.

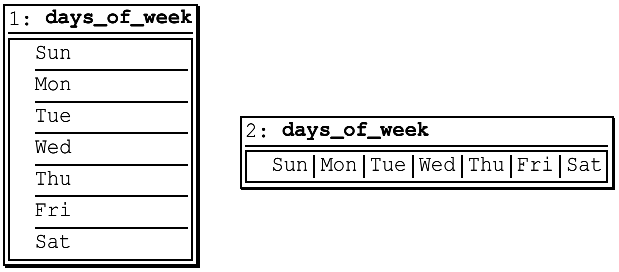

Figure 9: Simple arrays in DDD

Nested structures like multi-dimensional arrays are displayed by applying the `array()` function in a bottom-up fashion. First, `array()` is applied to the innermost structures; the resulting boxes are then passed as arguments to another `array()` invocation. The GDB output

```
{{"A", "B", "C"}, {"D", "E", "F"}}
```

representing a 2 $\times$ 3 array of character strings, is evaluated in VSL as `array(array("A", "B", "C"), array("A", "B", "C"))`, resulting in a horizontal alignment of two more alignments representing the inner arrays.

The actual DDD display is shown in figure 10. As demonstrated in the left display, the user can hide details of nested arrays to save space.

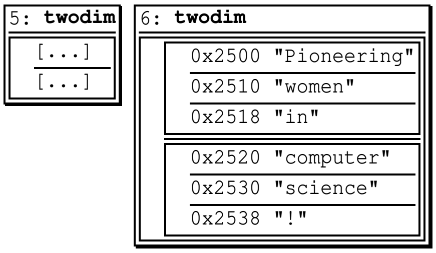

Figure 10: Nested arrays in DDD

Record structures are built in a similar manner, using a display function `struct_member` rendering the record members. Names and values are separated by an equality sign:

```
// Member of a record structure
struct_member (name, value) =
  name & " = " & value;
```

The display function `struct` renders the record itself, using the `valign()` function. `valign()` is similar to `halign()`, but builds a vertical alignment.

```
// Record structure
struct(...) = frame(valign(...));
```

Record structures, as displayed by DDD, are shown in figure 11. The actual VSL function takes care to align the equality signs; again, the user can hide the details of a nested structure.

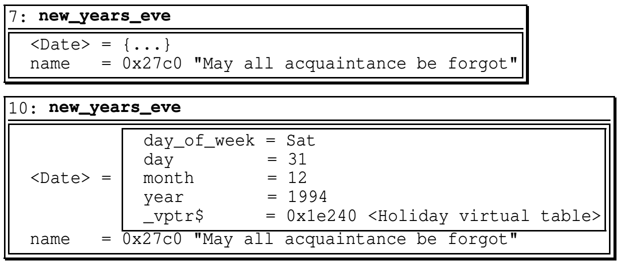

Figure 11: Record structures in DDD

6 Building Graphs from References

Using boxes is fine for displaying non-referential data structures—that is, data structures without references or pointers that can be denoted in a first-order term. However, referential data structures—i.e. data structures using references or pointers—can not be displayed using boxes. Instead, DDD displays a *graph* structure visualizing references as edges between the non-referential data nodes.

Whether data references other data is undecidable in general. Instead, the user decides about references when he creates a new data node. Each new data node can optionally be created as a *dependent node*, that is, with an edge originating from an already existing node to the new node. The by far most common operation to create a dependent node is to *dereference* a pointer value and thus creating a dependent node holding the value of the dereferenced pointer. In figure 12, an edge leads from the originating pointer value to the dereferenced value.

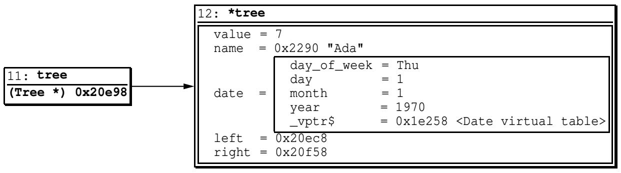

Figure 12: Dependent nodes in DDD

Besides simple pointer dereferencing, the DDD user may also modify the expression of the dependent node—to insert an additional type cast, for instance, or to visualize more complex references. DDD provides immediate creation of dereferenced nodes by a simple mouse click as well as dialog windows to enter and modify arbitrary dependent expressions. Since the resulting graphs may become large, the user can select, move and align nodes manually, but also invoke automatic graph layout procedures, as shown in the initial example.

7 Experiences

Writing a graphical front-end for GDB involved work at various levels. We had to implement a graphical display, where we could rely on the VSL library for easy creation of data displays and an existing graph editor. We had to code the DDD-GDB communication, where we could reuse an existing TTY-based interprocess communication package. The remainder was straight-forward programming of graphical user interfaces; after ten man-months of work, we had a full-fledged debugging environment. Only two additional man-months were required to make DDD generally available; most of the time was spent to adapt DDD to various platforms and implementing automatic configuration.

Today, DDD is the debugging environment we originally wanted: easy to use, with nice data exploration facilities, and relying on a powerful, robust, and freely available debugger, GDB. Besides fixing remaining bugs, our future work will focus on some enhancements to DDD. These enhancements, suggested by DDD users, shall include assembler support, an improved display structure merging displays with identical memory locations, and type-dependent displays allowing abstract views of abstract data types.[1]

As a conclusion, our experiences show that little distinguishes a well-written student implementation from a good software product; we wish to encourage academic institutions to enforce professional software standards from the very beginning and to promote sharing their efforts with others, as we did with DDD.

The DDD source code is available from several FTP servers around the globe. The FTP server of the X consortium, ftp.x.org, contains the most recent DDD source code in /contrib/utilities/ddd-*; please try a closer mirror site first. To build DDD, a recent C++ compiler such as GCC and a OSF/MOTIF library are required. DDD binaries for several platforms are available at the authors' FTP site, ftp.ips.cs.tu-bs.de, in /pub/local/softech/ddd/. Further information is found in the DDD WWW page, http://www.cs.tu-bs.de/softech/software/ddd_e.html.

[1] However, as all work on DDD is done on a volunteer base, contributions and donations for further DDD development are always welcome.

Acknowledgements. Several people contributed to the success of DDD. Above all, we want to thank the ever growing list of DDD testers, the first of them Carsten Krabiell and Petra Funk, whose support made DDD the tool it is today.

References

[1] CENTERLINE SOFTWARE, INC. *CodeCenter Tutorial and User's Guide*, version 4 ed. Cambridge, Mass., 1994.

[2] CHEUNG, P., AND WILLARD, P. *XXGDB – X Window System Interface to the GDB debugger*, Nov. 1994. Distributed with XXGDB.

[3] CYGNUS SUPPORT. *A Graphical User Interface for the GNU Debugger*. Mountain View, CA, Apr. 1995.

[4] HEWLETT-PACKARD, INC. *SoftBench Program Construction Tools—an Introduction*. Palo Alto, CA, 1992.

[5] KNUTH, D. E. *The TEXbook*. Addison Wesley Publishing Company, Reading, Massachusetts, 1984.

[6] LÜTKEHAUS, D. DDD – ein Debugger mit graphischer Datendarstellung. Master's thesis, Technical University of Braunschweig, Germany, Nov. 1994. In German.

[7] LÜTKEHAUS, D., AND ZELLER, A. *DDD – the Data Display Debugger*, version 1.2 ed. Technical University of Braunschweig, Germany, May 1995. Distributed with DDD.

[8] SCHUMACHER, M. *TGDB, a graphical frontend to GDB, the GNU debugger*. HighTec EDV-Systeme GmbH, St. Ingbert, Germany, 1994. Distributed with TGDB.

[9] STALLMAN, R. M., AND PESCH, R. H. *Debugging with GDB*, version 4.13 ed. Free Software Foundation, Jan. 1994. Distributed with GDB.

[10] SUN MICROSYSTEMS, INC. *Debugging Tools—DBX*, SunOS 4.1.1 ed. Mountain View, CA, Mar. 1990.

[11] ZELLER, A. VSE – ein generischer, visueller Struktureditor. Master's thesis, Technical University of Darmstadt, Germany, July 1991. In German.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 92-01

92-01	F.-J.Grosch, G.Snelting	Polymorphic Components for Monomorphic Languages
92-02	S.Conrad, M.Gogolla, R.Herzig	TROLL *light*: A Core Language for Specifying Objects
93-01	B.Fischer	A New Feature Unification Algorithm
93-02	G.Snelting	Perspektiven der Softwaretechnologie
93-03	G.Snelting, A.Zeller	Inferenzbasierte Werkzeuge in NORA
93-04	W.Rönsch, J.Schüle	Parallelisierung im Wissenschaftlichen Rechnen
93-05	P.Löhr-Richter, G.Reichwein	Object Oriented Life Cycle Models
93-06	M.Krone, G.Snelting	On the Inference of Configuration Structures from Source Code
93-07	S.Schwiderski, T.Hartmann, G.Saake	Monitoring Temporal Preconditions in a Behaviour Oriented Object Model
93-08	T.Hartmann, G.Saake	Abstract Specification of Object Interaction
93-09	G.Snelting, B.Fischer, F.-J.Grosch, M.Kievernagel, A.Zeller	Die inferenzbasierte Softwareentwicklungsumgebung NORA
93-10	C.Lindig	STYLE – A Practical Type Checker for SCHEME
93-11	H.-D.Ehrich	Beiträge zu KORSO- und TROLL *light*-Fallstudien
94-01	A.Zeller	Configuration Management with Feature Logics
94-02	J.Schönwälder, H.Langendörfer	Netzwerkmanagement — Beschreibung des Exponats auf der CeBIT'94
94-03	T.Hartmann, G.Saake, R.Jungclaus, P.Hartel, J.Kusch	Revised Version of the Modelling Language TROLL (Version 2.0)
94-04	A.Zeller, G.Snelting	Incremental Configuration Management Based on Feature Unification
94-05	S.Conrad	A Basic Calculus for Verifying Properties of Synchronously Interacting Objects
94-06	M.Gogolla, N.Vlachantonis, R.Herzig, G.Denker, S.Conrad, H.-D.Ehrich	The KORSO Approach to the Development of Reliable Information Systems
94-07	C.Lindig	Inkrementelle, rückgekoppelte Suche in Software-Bibliotheken
94-08	B.Fischer, M.Kievernagel, W.Struckmann	VCR: A VDM-based software component retrieval tool
95-01	V.S.Cherniavsky	Philosophische Aspekte des Unvollständigkeitstheorems von Gödel
95-02	G.Snelting	Reengineering of Configurations Based on Mathematical Concept Analysis
95-03	A.Zeller	A Unified Configuration Management Model
95-04	H.Bickel, W.Struckmann	The Hoare Logic of Data Types
95-05	F.-J.Grosch	No Type Stamps and No Structure Stamps – a Referentially-Transparent Higher-Order Module Language
95-06	V.S.Cherniavsky	Über semantische und formalistische Beweismethoden in den exakten Wissenschaften
95-07	A.Zeller, D.Lütkehaus	DDD - A Free Graphical Front-End for UNIX Debuggers