ALMA versus DDD

Daniela da Cruz[1], Pedro Rangel Henriques[1], and Maria João Varanda Pereira[2]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,prh}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

Abstract. To be a *debugger* is a good thing! Since the very beginning of the programming activity, debuggers are the most important and widely used tools after editors and compilers; we completely recognize their importance for software development and testing. Debuggers work at machine level, after the compilation of the source program; they deal with assembly, or binary-code, and are mainly data structure inspectors. To say that ALMA is a *debugger*, with no value added, is not true! ALMA is a source code inspector and deals with programming concepts instead of machine code. This makes possible to understand the source program at a conceptual level, and not only to fix run time errors.
In this paper we compare our visualizer/animator system, ALMA, with one of the most well-known and used debuggers, the graphical version of gdb, the DDD program. The aim of the paper is two fold: the immediate objective is to prove that actually ALMA adds value to *debuggers*; the main contribution is to recall the concepts of *debugger* and *animator*, and clarify the role of both tools in the field of *program understanding*, or *program comprehension*.

1 Introduction

In the context of our research on *language-based tools* and *program understanding*, we developed, some time ago, a program animator, ALMA, that has been confronted many times against debuggers; a common question is: "*what is the value added by ALMA to the usefulness of traditional debuggers?*"

ALMA was conceived in order to prove the potential of using a traditional compilers' approach to enable the construction of generic (algorithm/language independent) tools for program visualization/animation. That developing effort was justified because we did not find in the literature any tool of that kind; after reviewing the area of program animation, we have concluded that a generic visualizer animator (not specific of a family of algorithms or of a given programming language) was lacking. So it is difficult to compare ALMA with similar tools. After attaining that methodological/technological objective, it became evident that ALMA could also be used to help programmers and software-engineers in program understanding. This is possible because ALMA let us abstract from the

concrete syntax of programming languages and aids in associating (operational) meaning to programming concepts.

Our intention in this article is to claim and justify that *ALMA is not a debugger*! We start the paper, on one hand writing about *debuggers* (section 2) and introducing (subsection 2.1) one in particular (DDD), and on the other hand presenting ALMA (subsection 3.1) in the context of *animators* (section 3).

Then, in section 4, we compare ALMA versus DDD, presenting general arguments, and going through a concrete example—the analysis (i.e., the visual inspection) of the factorial program to comprehend the *recursive function-call mechanism*. To compare both tools, ALMA and DDD, we must study and confront many details. However, in this paper, we will focus just on those topics that we consider the most important: the *purpose* of the tool; its *architecture*; *how* and *what* is visualized.

2 Debuggers

In this section we intend to characterize the class of programs designated by *debuggers*. As we can find in the literature ([17, 14]) lots of things had been already written about debuggers. Instead of trying our own definition and characterization, we will present well-known and accepted descriptions.
According to many authors, and also found in the Webopedia[3], a first and very simple definition of *debugger* says that it is *a program used to find errors (bugs) in other programs, allowing the user to stop the execution of a given program at any point and examine and change the values of variables*. So, it is possible to list the three most important characteristics of a debugger:

- helps a programmer in finding bugs in programs;
- allows to inspect variable values;
- allows the use of breakpoints and step-by-step execution.

The Wikipedia[4] presents a more sophisticated and complete definition that corroborates the previous one:

> *"A debugger is a computer program that is used to test and debug other programs. ...*
> *When the program crashes, the debugger shows the position in the original code if it is a source-level debugger or symbolic debugger, commonly seen in integrated development environments. If it is a low-level debugger or a machine-language debugger it shows the line in the disassembly. ...*
> *Typically, debuggers also offer more sophisticated functions such as running a program step by step (single-stepping), stopping (breaking) (pausing the program to examine the current state) at some kind of event by means of breakpoint, and tracking the values of some variables. ...*
> *Some debuggers have the ability to modify the state of the program while it is running, rather than merely to observe it'".*

[3] URL `http://www.webopedia.com/` visited on June, 2008.
[4] URL `http://en.wikipedia.org/` visited on June, 2008.

The last statement emphasizes the important functionality that should be offered by any *debugger* concerning the possibility to hand the set of *variable-value* pairs.

Many programmers (especially those accustomed to IDE for software development) don't like to work with console debuggers. Nowadays, they use more sophisticated tools with navigation, visualization and animation features. As we can read in [14]:

> *"Visualization of computer algorithms, data structures, and program execution is a specific area in which visual tools can be used effectively to enhance the productivity of software development and usage. A visual debugger gives the user an opportunity to interact with the program graphically. The graphical display enhances the understanding of the specific way in which the program proceeds."*

Since ALMA is an animation system, it should be compared with modern debuggers with a visual and windows-based interface.

GVD[5], the GNU Visual Debugger now a component of GNAT (the GNU Ada IDE), could be an interesting choice. However, we chose DDD (Data Display Debugger) because it is one of the more sophisticated and well-known (more used) members of that kind of recent debuggers; an introduction to that tool, that will be further compared with ALMA (section 4), is presented in the next subsection.

2.1 DDD at a glance

The standard Linux debugger is GDB [9], the GNU debugger. GDB provides an interactive text-base method for accomplishing the tasks referred above, including step-by-step execution, breakpoints, variable watch and other options that are expected in a full-fledged debugger. GDB is in fact a tool still actively under development.

The last novelty relies on adding "reversible debugging" support [4] — allowing a debugging session to step backwards, much like rewinding a crashed program to see what happened.

To make easier the debugging of a source program, DDD [17] rose up as an X-Windows front-end to both GDB and DBX [7] debuggers. DDD *is just a front-end* — it does not do any debugging. Instead, it sends all user commands to a GDB (or DBX) running process, and limits itself to display the answer received from that back-end process.

Besides the usual FE features, such as handling *set and display* of breakpoints, DDD provides an interactive graphical data display, where data structures are shown as graphs. So, using DDD, we can follow the program execution visualizing the source code line-by-line and also watch its data structures.

[5] URL `http://directory.fsf.org/project/gnuVisualDebugger/`, or `http://www.lrz-muenchen.de/~Reinhold.Bader/ada_doc/html/gvd.html`, both visited on June, 2008.

Hence the DDD depends on GDB, and GDB depends on GCC, the Gnu Compiler Collection (previously, meaning Gnu C Compiler)[6], it is relevant to understand GCC's architecture, which is illustrated in Fig. 1.

Looking to the architecture of GCC we can remark that: the FE reads the source code and builds a parse tree; the parse tree is used to generate an RTL[7] instruction list based on the named instruction patterns; and the instruction list is matched against RTL templates to produce assembly or machine-code.

This architecture allow us to understand how GCC supports multiple languages—like Ada, Bash, C, C++, Chill, Fortran, Java, Modula, Pascal, Perl and Python; in fact, the FE parses each one and converts to RTL. Since RTL is a common intermediate representation (IR), very close to assembly language, this language is used by GCC to provide support to different languages. We additionally can say that to port the compiler to a new language is just needed to modify the FE; to alter the target architecture implies just a change on the BE.

Based on that kind of hierarchy (GCC $\longrightarrow$ GDB $\longrightarrow$ DDD), DDD is able to debug programs written in the languages supported by GCC (listed above).
It supports: machine-level debugging; hypertext source navigation and lookup; breakpoint, back trace, and history editors; program execution in terminal emulator window; debugging on remote host; GDB/DBX/XDB command-line interface with full editing, history, and completion capabilities.

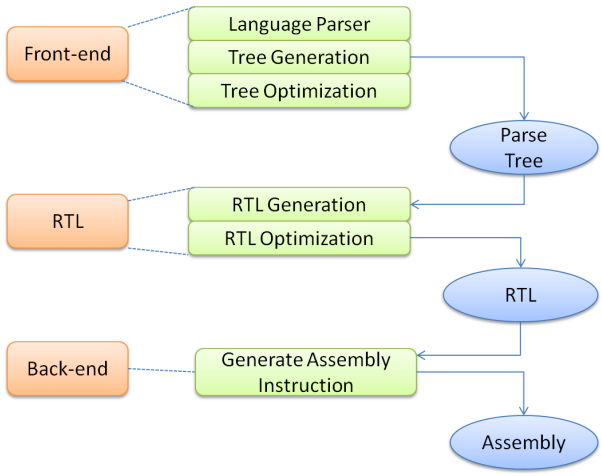

Fig. 1. Architecture of the Gnu Compiler Collection (GCC)

As can be seen in Fig. 3 (look at page 9), DDD environment consists of four windows:

[6] Remember that to run GDB on a program, it is necessary to compile before that program with GCC with `debugging option` activated.
[7] Register Transfer Language

– The "debugger window" which contains the actual communication between DDD and GDB;
– The "source window" which contains the source program and the basic source debugging action;
– The "command tool window" which contains buttons to activate most of the debugging actions (stepping up and down, setting breakpoints, etc);
– The "data window" which contains all data-related information, such as variable and function watching.

With all these powerful features, the debugging of a program becomes an easy task.

3 Animators

The role of the visualization technology, in fields like program comprehension and software engineering, is strongly recognized by the computer science community as a very fruitful feature. Indeed, the use of software visualization functionalities allows us to get a high quantity of information in a faster way.

The purpose of an *animation system* is to construct (automatically) program visualizations in order to help the programmer *to inspect data and control flow* of a source program and *to understand its behavior*. This kind of information can be used by the programmer to recognize: the main tasks performed in the program; the changes in the value of variables; or possible errors. It also can be used to verify the correctness of the program results and *their meaning*. The values computed by a program have an effect over the object under control; so it will be also desirable to visualize the program effects, this is the meaning of the program.

In an animation system, a special care should be taken with content (subset of program information that will be visualized). It is also necessary to decide: how this content will be visualized; what kind of visualizations will be used; what kind of interaction facilities should be made available; how to solve scalability problems; the colors/sound usage, etc.

In an animation system, there are several kinds of views that can be produced: *operational* (program level) or *behavioral* (problem level); *static* (compile-time) or *dynamic* (runtime); *structural* or *quantitative* (based on metrics or other kind of statistical information).

Animation systems usually are designed to include some features that are not available in debuggers. While animation systems are strongly related to program comprehension and cognitive models, in the context of machine-level debugging that makes no sense.

Usually, in animation systems the main research concerns are visualization techniques: to adapt to the knowledge level of the user; to provide multiple views; to map one view to another; to support learner-built visualizations; to complement visualizations with explanations; to support different abstraction levels, etc. These features are essentially *concept-oriented*. In debuggers, the concerns are: how to enable forward or backward traversal of the execution path; how to

report historical information; how to support user's input data; how to report one variable life cycle; how to use dependency graphs, etc. These kind of features are essentially *operation-oriented*.

In the next subsection we present the ALMA system, explaining briefly its architecture, how it supports different source languages, and the animation process.

3.1 Alma at a glance

After reviewing the existing systems (like BALSA([2] in [12]), TANGO ([13], PECAN ([10] in [12]), FIELD ([11] in [12]), JELIOT [5] and CENTAUR [1]), we decided to design ALMA aiming at being a new generic tool for program visualization and animation based on the internal representation of the input program in order to avoid any kind of annotation of the source code (with visual types or statements), and to be able to cope with different programming languages.

To fulfill such requirements, we had been inspired in the classic structure of a compiler and we conceived an architecture that separates the source program recognition from its animation, using a decorated abstract syntax tree (DAST) as internal representation (see Fig. 2).

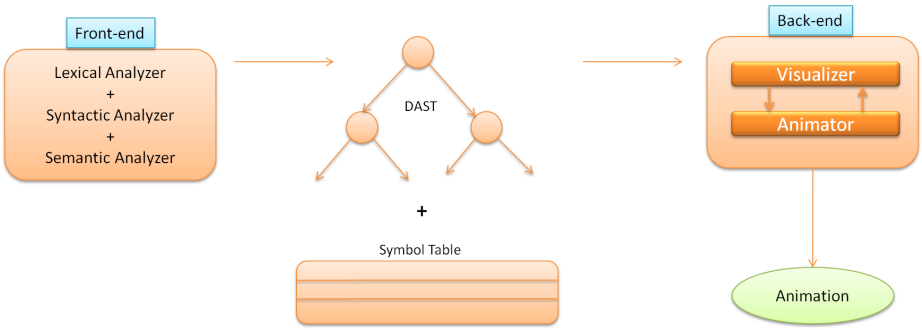

Fig. 2. Architecture of ALMA system

ALMA was implemented in Java, using and reusing the compiler generator system LISA [8], as specified and described in [15], [6] and [16].

ALMA system has a front-end (FE) specific for each language and a generic back-end (BE). It uses a decorated abstract syntax tree (DAST) for the internal representation of the program's meaning; it is the connection between the FE and the BE. Using a DAST as an internal representation, we isolate all the source language dependencies in the FE, while keeping the generic animation engine in the BE. The DAST is built using a set of pattern rules [3]. A pattern represents an abstraction of a programming concept. Each one of these patterns is composed by two parts: a structural component (given by a grammar production) and

a semantic component (given by a set of attribute occurrences affected to the symbol that labels each tree-node).

These patterns capture the abstract syntax of each entity (value or operation) in order to preserve and keep, via attributes, the necessary information to express its static semantics. So the DAST do not reflect directly the source language syntax, and in this case, the tree generated by the FE (after parsing the source) represents the input program.

Applying specific rewrite rules (which are used according to the pattern-tree found in the DAST) to the *execution tree*, we obtain a description of the different program states, simulating its execution.

A *Tree-Walker Visualizer*, traversing the *execution tree* and applying visual rules create a representation for the nodes generating a visualization of the *program tree* in that moment. Then the DAST is rewritten (to obtain the next internal state), and redrawn, generating a new visualization which reflects the new state of the program.

This approach, using a DAST as internal representation and a set of pattern rules allows us to easily construct different abstraction levels of the same program since the operational view till the behavioral view. For that, it suffices to associate a new set of rewrite and visualize rules to the DAST patterns. This system is based on the concepts involved in a program and not directly in the source code.

As can be seen in Fig. 5, ALMA's interface is split in four windows—3 main windows and 1 with the buttons for navigation—with the following content:

- The identifier table (on the top, to the left);
- The source text of the program to be animated (on the bottom, to the left);
- The program tree (to the right, occupying most of the display);
- The interaction buttons that allow the user to control the system. Using `back` and `forward` controls the user will be able to navigate through the program animation, step by step.

Colors are used to make easier to follow the animation of a program: Red color indicates the identifier / subtree that will be changed by the next execution step (the execution of the program-statement also colored in red); Green color indicates the new status of the program (subtree /identifier) after the last change.

In the next section we finally show how different is ALMA from DDD. For that purpose a simple source program written in C—the recursive implementation of the *factorial function*—is used. This example, although simple, is short and allow us to focus in the most important and crucial details.

4 Alma vs DDD

To do the comparison between ALMA and DDD, we will show how each one of the tools holds and shows the information for the same source program listed below—a C program that implements the *factorial function*.

After loading *factorial program*, DDD displays the information illustrated in Fig. 3.

```
1  #include <stdio.h>
       int factorial(int n) {
3          int res = 1;
           if (n > 0) {
5                res = n*factorial(n-1);
           }
7          return res;
       }
9      int main() {
           int a, r;
11           scanf("%d",&a);
             r = factorial(a);
13           printf("%d",r);
           return 0;
15     }
```

Our main purpose, exploring this example, is to focus on the recursive mechanism: How does DDD deal with the function-call process? How does it cope with the parameter-passing process?

Suppose that we add a breakpoint to the program at line 5 (recursive call). Figures 3 and 4 illustrate what happens when, evaluating the factorial of 4, we execute 2 steps (selecting `Step` action twice), after DDD stops at the breakpoint. Observing those figures, we see that: in the stack the double call of `factorial` function is explicit, first with the value 4, and then with the value 3; the next line executed is the line 4; and this is also visible in the frame and in the registers table.

But, if we are trying to understand what values are actually passed and how they are assigned to the function parameters, DDD is not of a big help, because it just displays the values of the arguments when the function execution starts (after the assignment).

If we do not know assembly language at all, the use of a debugger is very complex or not really helpful. In fact, if we are just interested in understanding the program's control or data flow, the visualization of that mess of registers, and hexadecimal codes or addresses can be awkward.

If this is the case, the use of ALMA can be helpful. Following the previous procedure (above done for DDD), we can observe in figures 5, 6, and 7 the output of ALMA for the function-call, parameter passing, and function execution.

Each time a function is called a new window is opened, being drawn in a cascade style (see Fig. 6). As the new function has its own memory area, a new window shows the function execution environment (IdTable), i.e., the local and global identifiers; the program-tree for that function is also displayed.

Looking at Fig. 6, we see that the call of *factorial* function causes a jump to the function-header to show how the parameters are passed—this is a completely different approach of the one followed by DDD. Notice that the identifier table, during the argument-passing phase, is different from the identifier table available during the execution of the function body (Fig. 7). At this moment (argument-passing), the active identifier table is the one belonging to the calling function plus the function arguments (defined in the *factorial* header).

After passing the parameters to the function, a new sub-window—named `Executing function`—opens at the right of the previous one to display the tree

corresponding to the function body (Fig. 7) and to show its execution. At this phase, the identifier table is the one strictly accessible inside the called function (including just the global variables and those variables local to *factorial*).

It is evident that ALMA displays more information, and in a clearer way, than DDD for the same execution process: the function/procedure call (recursive or not).

Another interesting/important case to compare DDD and ALMA is the visual inspection of structured data types. For that purpose, we will consider the visualization of *arrays*.

In DDD, displayed pointer values are dereferenced by a simple mouse click, allowing to unfold arbitrary data structures interactively. DDD automatically detects if multiple pointers refer to the same address and adjusts the display accordingly. DDD has a major drawback: each and every pointer of a data structure must be dereferenced manually. While this allows the programmer to set a focus on specific structures, it is tedious to access, say, the 100th element in a linked list.

In ALMA, each element of a structured data type is displayed in a row of a table, freeing the user from the tedious task of clicking in each element to discover the next one. If the array is larger, the user only needs to scroll down.

These situations are illustrated in Fig. 8 and 9 for DDD and ALMA, respectively.

So, we are able to say, after comparing DDD and ALMA tools, that ALMA provides a more effective aid in the context of program comprehension/understanding.

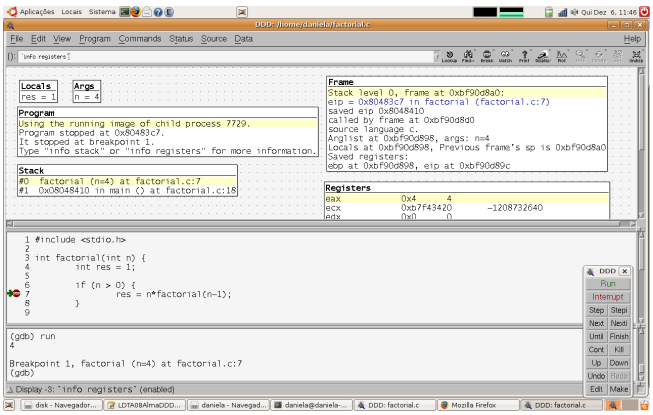

Fig. 3. Executing the factorial function in DDD — recursive call (1)

Below are the stronger arguments, that came out from the comparative study, to affirm that *definitely ALMA is not a Debugger.*

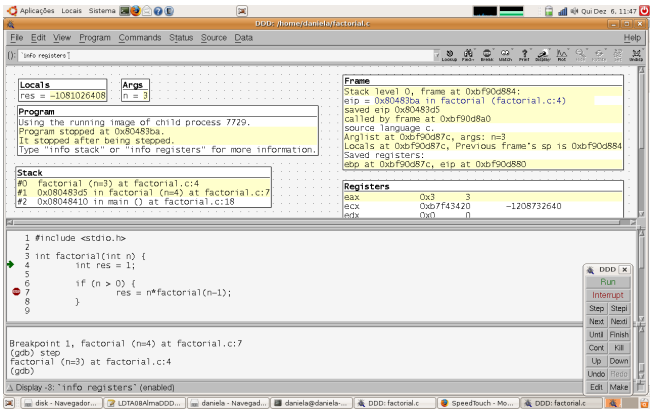

Fig. 4. Executing the factorial function in DDD — recursive call (2)

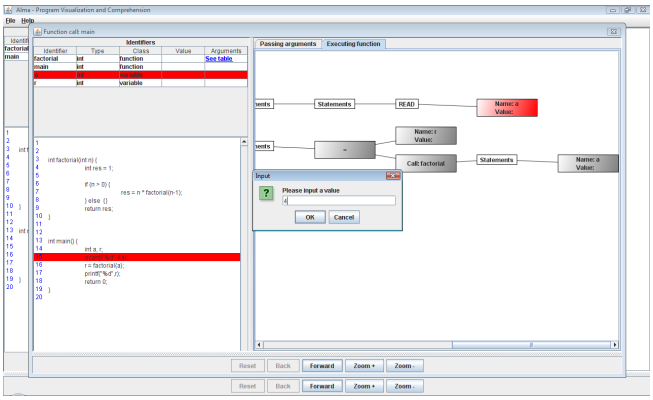

Fig. 5. Executing the *main* function in ALMA— reading a variable

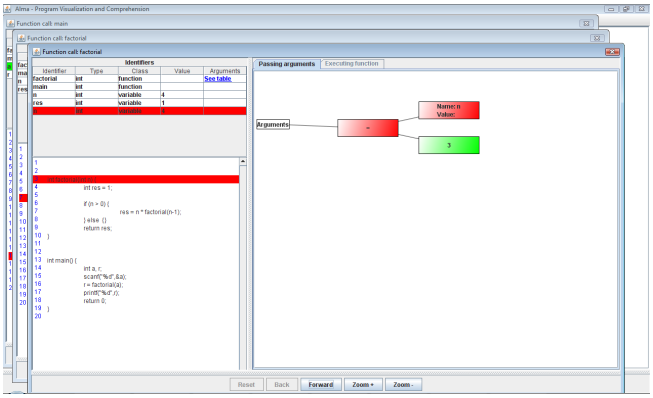

Fig. 6. Passing parameters to the *factorial* function in ALMA

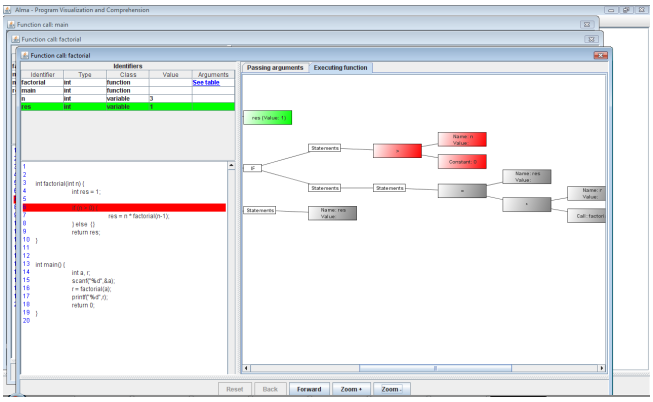

Fig. 7. Executing the body of *factorial* function in ALMA

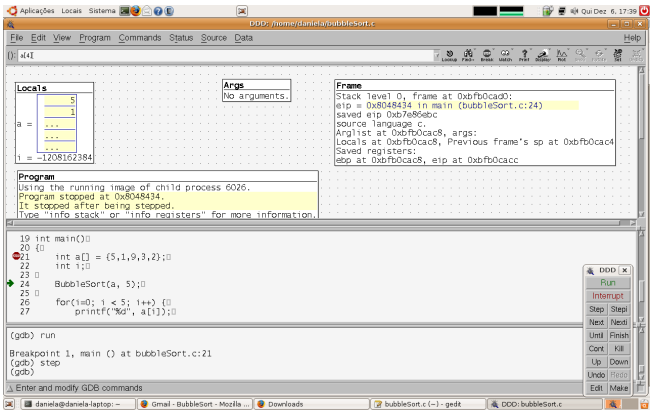

Fig. 8. Display of the elements of an array in DDD— at the top left

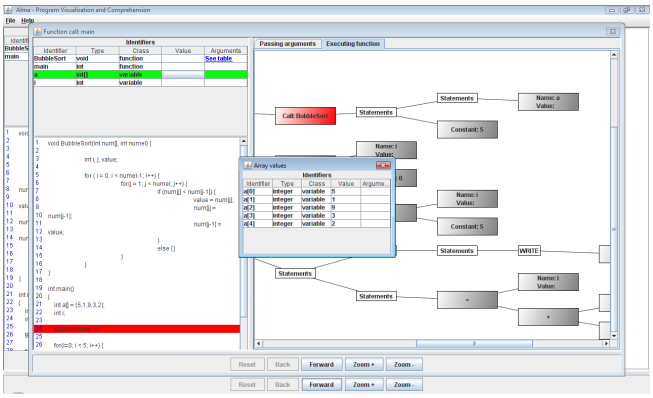

Fig. 9. Display of the elements of an array in ALMA

- ALMA is not a traditional (low-level) debugger in the common sense of a tool that helps the programmer to fix errors in the assembly/machine-code. Alma creates a visualization for abstract programming concepts and animates them. ALMA can simulate the source code execution (on an operational visualization level) but it does not compile the source program, neither it reuses the assembly code.
- ALMA is not C/RTL-oriented, it is language-independent.
  Even knowing that DDD can cope with any language $\mathcal{L}$ that compiles to RTL it is much more difficult/complex to create a compiler to RTL then a $\mathcal{L} - FE$ for ALMA (mapping $\mathcal{L}$-statements to abstract high-level concepts).
- DDD works at an *operational-level* (machine level), while ALMA can produce visualizations at a *concept-level* (programming abstract level) or at an operational level as we presented in this paper. It depends on the visualization rules associated to the program internal representation.
- ALMA is aimed at explaining (illustrating) the program semantics that can be useful to detect/understand algorithmic errors, while DDD and other debuggers are oriented to low-level error detection.

Notice the way ALMA deals with parameter passing referred above. In any high-level language this mechanism is entirely embedded in the subprogram invocation, and the underlying idea of assigning values (the actual parameters) to the local variables (the formal parameters) is not explicit anywhere. As long as ALMA works with concepts, this mechanism is made explicit...

Of course ALMA has some limitations that most of the debuggers do not have—ALMA is an academic prototype, while the other are professional, or commercial, tools. ALMA does not support at this moment *pointers* neither *objects*, and is much poor concerning the interaction with the user during the animation (no breakpoints available, etc.). To deal with multiple language complex applications, although not experimented, we defend that it will not be a problem, as it is a matter of converting the various modules to the DAST uniform representation.

5 Conclusion

ALMA and DDD (or any other visual debugger) are very similar, at least at a first sight, and many people says that ALMA is just a new debugger without any value added. Because of that we decided to carry on a study aiming at gathering arguments to agree or refute that statement. For that purpose, we selected DDD to confront with ALMA. This article reported that work, concluding that ALMA is not a debugger! After a careful study of both tools (just summarized here), we went through an example—the recursive computation of the factorial—highlighting the points of the visual inspection offered by each tool that support the arguments presented at the end of section 4.

References

1. Yves Bertot. Occurences in debugger specifications. In *PLDI91*, 1991.

2. M. H. Brown and R. Sedgewick. A system for algorithm animation. In *SIGGRAPH'84*, volume 18, pages 177–186, Minneapolis, July 1984. ACM Computer Graphics.
3. Daniela da Cruz, Maria João Varanda Pereira, and Pedro Rangel Henriques. Constructing program animations using a pattern-based approach. 2, December 2007.
4. GDB. Gbd and reverse debugging. http://sourceware.org/gdb/news/reversible.html, 2007.
5. J. Haajanen, M. Pesonius, E. Sutien, T. Terasvirta, P. Vanninen, and J. Tarhio. Animation of user algorithms in the web. In *VL'97 - IEEE Symposium on Visual Languages*, pages 360–368. IEEE, Setembro 1997.
6. Pedro Henriques, Maria João Varanda, Marjan Mernik, and Mitja Lenic. Automatic generation of language-based tools. In *LDTA - Workshop on Language, Descriptions, Tools and Applications (ETAPS'02)*, April 2002.
7. Mark A. Linton. The evolution of dbx. In *USENIX Summer*, pages 211–220, 1990.
8. Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system LISA. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.
9. R. Pesch R. Stallman and S. Shebs. *Debugging with GDB - The GNU Source-Level Debugger*. Free Software Foundation, 2002.
10. Steven Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software engineering*, 1985.
11. Steven Reiss. Interacting with the FIELD environment. *Software Practice and Experience*, 1990.
12. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1997.
13. John T. Stasko. Simplifying algoritm animation with TANGO. In *IEEE Workshop on Visual Languages*. IEEE, Outubro 1990.
14. Dan E. Tamir, Ravi Ananthakrishnan, and Abraham Kandel. A visual debugger for pure prolog. *Inf. Sci. Appl.*, 3(2):127–147, 1995.
15. Maria João Varanda and Pedro Rangel Henriques. Visualization / animation of programs based on abstract representations and formal mappings. In *HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*. IEEE, September 2001.
16. Maria João Varanda and Pedro Rangel Henriques. Visualization / animation of programs in alma: obtaining different results. In *VMSE2003 - Symposium on Visual and Multimedia Software Engineering (HCC'03), New Zealand*. IEEE, October 2003.
17. Andreas Zeller and Dorothea Lütkehaus. Ddd - a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.