Visualizing Dynamic Data Dependences as a Help to Maintain Programs

Françoise Balmas Harald Wertz Rim Chaabane
Laboratoire Intelligence Artificielle
Université Paris 8
93526 Saint-Denis (France)
{fb,hw,lysop}@ai.univ-paris8.fr

Abstract

This paper is on a project to evaluate the impact of visualizing dynamic data dependences in the context of maintenance activities.

Our work is based on previous work in displaying static data dependences and on experience with large sets of dependence displaying strategies that we adapted to deal with problems where dynamic information is crucial. We developed a prototype around a Lisp interpreter and applied it to a highly complex AI program. This permitted us to build efficient visualizations and to evaluate the benefits of using dynamic dependences for program understanding, debugging and correctness checking.

In this paper, we present our prototype, detailing especially the different visualizations we introduced to allow users to deal with hard to understand programs, and we discuss how dynamic dependences permit to see *what really happens during program executions.*

1. Introduction

In the past, we used static data dependences to help understand and document programs [1], and developed displaying strategies to deal with large sets of dependences [2]. In this context, we showed that visualizing sample values of variables, for a well chosen execution, was very efficient to help understand what a program does and how it works. That's why we decided to explore computing dynamic dependences and to evaluate the benefits of visualizing them for those activities where knowledge about given executions is crucial, that is program understanding, debugging and correctness checking.

For the sake of evaluation, we developed a prototype around the Lisp language; actually, modifying an interpreter is much easier than modifying a compiler, and hard to understand Lisp programs are still small enough to prevent algorithmic and optimization problems which arise when manipulating huge amounts of data.

To evaluate our approach, we applied our tool to a version of the classical AI Blocks World program [4]. In our version, the world is a table with different objects on it which can be manipulated by a one-handed robot. Basically, the program presents itself as an interpreter the user interacts with in order to create objects, make the robot move them to other places or ask for information about the current state of the world.

The program is around 1200 LOC long[1] and includes more than 125 functions and macros, many global variables modified through pointers, indirect recursive calls, thus long circularities, and escapes (i.e. non standard return controls). It evolved over time, since first developed for teaching purpose and then modified several times to add further reasonning capabilities. All these features make this program rather complex, hard to understand for newcomers to the program and difficult to maintain for the one of us who developed it.

In this paper, we report on this evaluation, discussing both the different kinds of visualizations we defined and the way they let us *see* what happened during execution of our program, helping us to understand, debug and check it for correctness.

2. The tool

Our tool relies on three modules: a modified Lisp interpreter (a C version is under construction), a database (currently a Lisp program) and a GUI (implemented in Tcl/Tk). We modified a Lisp interpreter to make it, in addition to normal execution of programs, extract dependences at runtime. These dependences are sent to a Lisp program that acts as a database, storing the dependences and producing,

[1] Note that LOC in Lisp is very different from LOC in more usual programming languages such as C, because of the compactness of code and the powerfull functional primitives it offers.

```
(de square (a)
   (* a a))

(de som2 (x y)
   (+ (square x) (square y)))

? (som2 3 5)
= 34
```

Figure 1. Sample code

on demand, the corresponding graph – in *dot* [3] format. Finally, a Tcl/Tk GUI displays the graph, using mechanisms to reduce its size, and allows users to interact with it to tune several kinds of visualizations.

The full set of dependences for a given call is unlikely to be displayed as is, since it is usually to large to be readable. We thus *group* together nodes (that is pieces of code) belonging to the same function call. For example, in the sample code of Fig. 1, wich computes the sum of the square of two numbers, we have nodes belonging to the two calls to function square and we aggregate them to form two groups. These two groups, as well as other nodes, belong to function som2 and are aggregated to form the main group. We can then display dependences showing only these groups, thus only the calls, and the dependences between them. Fig. 2 gives the corresponding graph for the call (som2 3 5) and shows how values are transmitted between calls. Alternatively, we can also get a graph with only the toplevel call visible (see Fig. 3), showing just input and output of the whole program. Such views are very helpful when global variables are used and modified by the program (see Section 4).

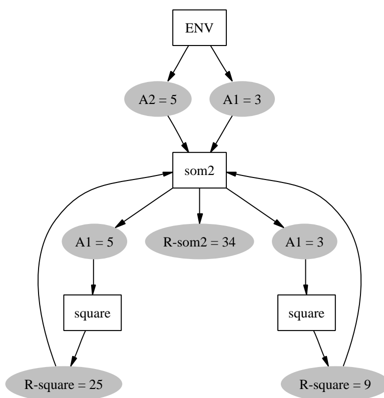

Figure 2. Data dependence graph with all calls visible

For a large program, the number of function calls may become also too large to get readable graphs. For this reason, we introduced a tagging mechanism to classify functions into control structures (they are functions in Lisp), primitives (those standard functions that are implemented in Lisp itself) routines (small reusable functions related to the program at hand) and user functions (all the remaining functions). The next Section will show different visualizations that depend on this classification to filter out given set of calls.

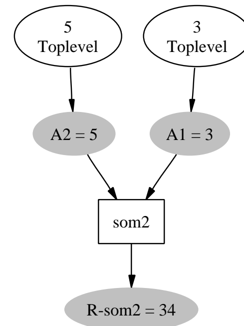

Figure 3. Data dependence graph with only the toplevel call visible

3. Visualizations

In this section, we introduce the different visualizations our tool offers to help users analyze programs. The first four are variants of call graphs: we noticed that navigation inside data dependences graphs is often tedious and call graphs provide a good 'map' to support this navigation. The last four visualizations are variants of data dependence graphs.

Call graph Such a visualization offers a global overview of the functions the program evaluted and the way they are organized. It also permits the user to ask for a given data dependence graph by interactively selecting a call: this call then becomes the *focus* of the displayed data dependence graph (see below).

User call graph This is a restricted version of the call graph just described, where only user functions are shown. This permits to get a graph with much fewer calls – from more than 3600 calls in the whole call graph for a 'move-object' instruction to our robot we could get down to about 30 calls –, thus more easily readable. This also permits to get a global overview of the main function calls from a programmer's conceptual perspective.

One level user call graph This visualization is a mix of the two previous ones: a call graph beginning at a given user

function and ending at the next call of a user function. That is: when traversing the call tree, we stop drawing the graph when we reach leaves or we encounter user functions. This visualization gives all necessary details but locally bounded by user functions.

Return graph The Blocks World program uses intensively the 'escape' mechanism of Lisp[2] that allows the program control to directly return to a calling function up in the call tree. It is then often hard to conceptually follow where the control is supposed to get back and how the program is supposed to continue after the activation of the 'escape'. That's why we integrated the possibility to extend the *call* graphs with the *return* graph: whenever control gets back to another function than the one that called the current one, the return arrow is displayed in red.

Data dependence graph This visualization provides the standard data dependence graph we introduced in Section 2, with either only the toplevel call, or all calls. It may focus on a given call, this way considering only the sub tree beginning at this call.

Filtered data dependence graphs This visualization is obtained whenever classes of functions are tagged to be filtered out. It is especially useful with data dependence graphs where all calls are to be displayed, since it permits to hide functions of lesser interest for the task at hand. For example, to focus on the dependences from a programmer's conceptual perspective, it is useful to filter out control strucutres, primitives and routines that often fill a graph with irrelevant information.

First level graphs The two basic possibilities to examine calls – only the toplevel call visible, or every call visible – proved to be insufficient in several cases, since giving either too few or too many details. We extended our tool functionnalities with a view where the function call focused upon is visible along with each first level call. This allows the user to examine how a given action – implemented by a function call – is decomposed into smaller actions, without the need to examine the actual code of the call.

Sets of calls Sometimes, the automatically built views we just described are not satisfying because centered on *one* call, while we might need the ability to see a *set* of specific calls, especially to examine the values of global variables before and after these different calls (see Section 4). For this reason, selecting a few calls on a call graph results in a data dependence view where only these calls are shown while all others are hidden.

[2]Sometimes called 'catch-and-throw', this mechanism is similar to the 'setjmp-longjmp' mechanism of C.

The different visualizations presented in this section were inspired by the needs we encountered during the process of trying to understand and evolve a rather large and complex program. They showed to be very useful for interactive goal-directed exploration. In the next Section we will discuss more specifically the use of dynamic data dependences for different programming activities.

4. Dynamic data dependences for programming activities

Program discovery The first context where our visualizations proved to be useful is program discovery, that is the task a programmer faces when s/he has to get aquainted with a program s/he didn't implement her/himself. Even if interacting with the robot, on the Lisp terminal, was easy to grasp, trying to understand how the program works in order to handle object creation, placement and moving was another question!

A data dependence graph focused on the toplevel call is a good view to start with, since it shows how global variables are modified during the call. For example, Fig. 4 shows how the table, the object list and the object itself are modified during the creation of an object, showing this way the real effect of the call. The user-call-graph permitted us then to get a global overview of the actions performed, while tuning data dependence graphs for these different actions gave us futher information on how they affect the global variables.

Finding bugs While working on the discovery of the Blocks World program, we encountered graphs with wrong variable values, incorrect number of function calls or unexpected calls. Refining different graphs, we could navigate backward and forward to find the source of the problem.

For example, we noticed that when finding a place where to put a square object of 2x2, the robot only checked three positions on the table, while of course it should have checked at least four. This example shows that our visualizations may direct users to problems they don't even suspect: we didn't *search for* any problem in finding a place, we just *saw* there was one. We could then detect more precisely why there was this problem and how to solve it.

Correctness checking As an extension of the two former points, we also used our views to verify that the program was behaving properly. For instance, careful inspection of visualizations of the program's execution after correction of the 'finding place' bug permitted us to *see* its correctness.

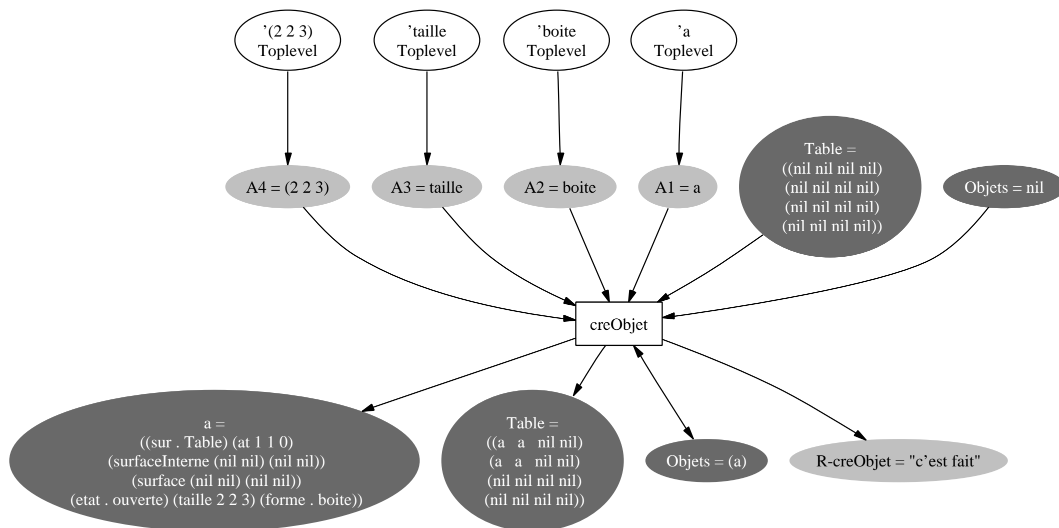

Figure 4. Overview of computation performed

We also used our views to verify that the program was behaving the way we expected it to do. As usually in AI programs, in many contexts large parts of the program – a function along with every call it performs – are reused and reused again, resulting in deep and broad call trees, extremely difficult to capture. To check that such functions were correctly implemented, we looked at user call graphs to check whether they were recursively called the correct number of times. We also looked at data dependence graphs where we rendered visible only calls to these functions, to examine the values of the global variables at the different steps of the program execution and to verify that they were modified the way we expected.

5. Conclusion

From our experience working with the Blocks World program, as well as several other programs, we can affirm that the major benefit given by the dynamic dependences our tool handles is that precise information about a program execution is recorded and visible, after execution, for examination: details about how execution was driven from one expression to another, as well as about the values variables had at any point of the program and how these values are transmitted from point to point.

The different visualizations we propose were designed to minimize the conceptual overload in order to allow users to *see* the exact information they need, otherwise barely accessible in the database. Clearly, dynamic information is of great help when working on problems like debugging, verifying that a program works properly, or even optimizing, since it gives information only for *one* given execution, when static dependences would give too much information.

On the other hand, the weakness of this approach is that it requires enough knowledge from the user on the possible paths in the programs: verifying that a program behaves properly means checking *many* possible executions, and the user has to find which ones are necessary. However, our approach also makes possible to discover some unforseen execution pathes, as static analysis would. Combining static information with dynamic dependences is one of our main perspectives. Two other perspectives are to extend our filtering mechanism to global variables – when they are not of interested at the same time, filtering out some would be useful – and to develop a query language permitting us to find, thus to jump to, parts of the execution corresponding to given criteria.

References

[1] F. Balmas. Using dependence graphs as a support to document programs. In *Proceedings of the Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, 2002.

[2] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal on Software Maintenance and Evolution: Research and Practice*, 16(3):151 – 185, May/June 2004.

[3] E. Koutsofi os and S. North. *Drawing graphs with* dot. AT&T Labs – Research, Murray Hill, NJ, March 1999.

[4] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.