

DDgraph: a Tool to Visualize Dynamic Dependences

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

Abstract

Following previous work on displaying static data dependences and experience with large sets of dependence displaying strategies, we developed a tool for visualizing dynamic data dependences.

Our prototype is based on a modified Lisp interpreter and this paper presents our evaluation of its application 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 our findings working with dynamic dependencies.

1. Introduction

In this paper, we report on our research using dynamic data dependences during program maintenance.

In previous work on static data dependences [3], where we developed displaying strategies for very large sets of dependences, we discovered that visualizing sample values for a well chosen execution could be of great help to understand what a program computes and how it works [2]. This pushed us to explore dynamic dependences – dynamic analysis is recognized to bring *precise* information for a given execution [1] – 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. We thus modified a Lisp

interpreter to let 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 on demand the corresponding graph – in *dot* [5] format. Finally, a Tcl/Tk GUI displays the graph, using strategies to reduce its size, and allows users to interact with it to tune several kinds of visualizations.

To evaluate our approach, we applied our tool to a version of the classical AI Blocks World program [6]. In our version, the world is a table and the blocks – different possible shapes of objects – are manipulated by a one-handed robot. Basically, the program presents itself as an interpreter the user interacts with in order to create objects, let the robot move them to other places or ask for information about the current state of the world.

The program is around 1200 LOC long¹ 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 an AI programming class and then modified several times to add further reasoning 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 present our tool (Section 3), the different kinds of visualizations we defined (Section 4) and then we discuss the benefits we got for the maintenance of a hard to understand program (Section 4 and 5).

2. 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

¹Note 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

normal execution of programs, extract dependences at run-time. These dependences are sent to a Lisp program that acts as a database, storing the dependences and producing, on demand, the corresponding graph – in *dot* [5] 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 too large to be readable. For this reason, following our past experience with displaying strategies to deal with large sets of dependences [3], we integrated *aggregation* and *filtering* mechanisms in our tool.

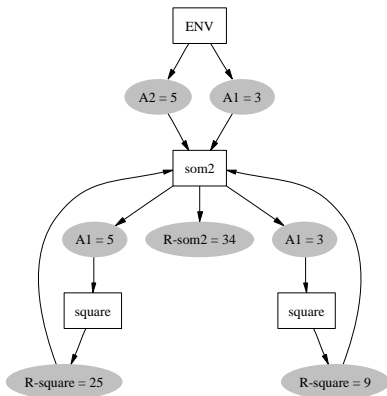


Figure 2. Data dependence graph with all calls visible

Aggregation is done by *grouping* together nodes (that is pieces of code) belonging to the same function call. For example, in the sample code of Fig. 1, which 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

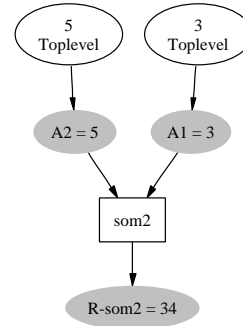


Figure 3. Data dependence graph with only the toplevel call visible

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).

For a large program, the number of function calls may become too large to get readable graphs. For this reason, our filtering mechanism classifies 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.

3. Visualizations

Our basic navigational functionalities – going down/up one level while opening/closing groups – becomes tedious as soon as the call tree exceeds more than a dozen levels. Actually, a typical call to the robot instruction for moving an object produces a call tree of more than 3600 groups (calls), distributed in a maximum depth of 90 levels and 45 in the mean. That’s why we propose different visualizations of the call graph to use as an help either to understand the program or to navigate in the dependences graphs. This Section introduces the different possible visualization of both call graphs and data dependence graphs.

Call Graph This view is based on the group hierarchy created to handle aggregation and shows the different calls performed during the program execution. It is displayed in another window than the data dependence graph.

Such a visualization offers a global overview of the functions the program evaluated and the way they are organized (see the Section 4). It also permits the user to ask for a given data dependence graph by interactively selecting a call. This

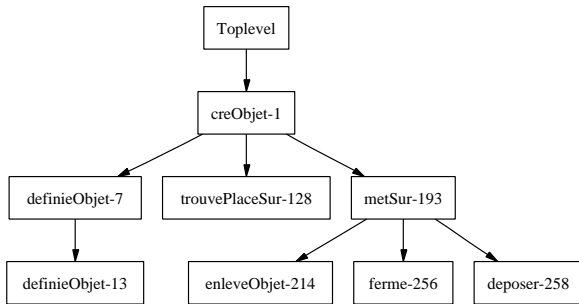


Figure 4. User call graph

group becomes then the focus of the displayed data dependence graph (see 3).

Note that such call graphs may be very large, thus restricted versions are also available (see below).

User call graph This is a restricted version of the call graph just described where only user functions are shown. This not only permits to get a graph with much fewer groups – from more than 3600 groups in the whole call graph for a ‘move-object’ instruction we could get down to about 30 groups –, thus more easily readable, but also to get a global overview of the main function calls from a programmer’s conceptual perspective. In Fig. 4, we see the user call graph for the creation of an object: from the initial 159 groups, only 9 are displayed.

One level user call graph This view is a mix of the two previous. Actually, in many cases, once the programmer found the function s/he is interested in investigating further, s/he might be willing to know more about *all* the calls performed by this function, and not only the user function calls. For this, we provide a call graph beginning at a given user function and ending at the next user function call, that is when traversing the call tree, we stop drawing the graph whenever we reach leaves or we encounter user functions.

Return graph The Blocks World program uses intensively the ‘escape’ mechanism of Lisp² that allows the program control to directly return to a calling function up in the call tree, restoring the local environment of the place where the ‘escape’ was set. If this clearly eases coding and speeds up execution time – less tests are to be written and evaluated – it also seriously complicates maintenance and debugging: as soon as several ‘escapes’ are embedded, because in recursive calls, it becomes hard to conceptually follow where the control is supposed to get back and how the

²Sometimes called ‘catch-and-throw’, this mechanism is similar to the ‘setjmp-longjmp’ mechanism of C.

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 as we introduced in Section 2. It follows the global setting of group display: each group can be open or closed, and visible or hidden when in a closed group. Furthermore, it may focus on a given call, this way considering only the sub calltree beginning at this call.

The construction of several different views is possible. When all groups are visible, the visualization gives a global overview of the different calls of a program execution, showing more specifically how arguments and returned values are transmitted between calls. When only the main group is visible, one can clearly see the effect of the call on global variables. When one or more groups are open, examination of the detail of the evaluated code is possible.

Examples are given in Sections 2 and 4 (Fig. 2, 3, 5 and 6). The next Section will further discuss this visualization.

Filtered data dependence graphs This visualization is obtained whenever classes of functions are flagged to be filtered out. It is especially useful with data dependence graphs where all groups are to be displayed, since it permits to hide functions of lesser interest for the task at hand. For example, it is often useful to filter out primitives – very often recursive functions called a huge number of times – that fill a graph with irrelevant information. Displaying control structures is also often useless when the programmer is more interested in focusing on *what* the program computes than on *how* it does it. To the contrary, s/he might be interested in examining the overall control of the program execution without considering how it is encoded in functions.

With this mechanism, one has just to tune the settings for each class of functions and then to select a group – in a call graph for example – and the tool automatically builds the corresponding view.

First level graphs The two basic possibilities to examine groups – only the top group visible, or any group visible – proved to be insufficient in several cases, since giving either too few or too many details. We extended our tool functionalities with a view where the focus group is visible along with each first level group. 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, which is always visible through

the group nodes. One can then navigate up/down one level for further examination.

Note that the filtering out of given function classes is also active in this view.

Sets of groups Sometimes, the automatically built views we just described are not satisfying because centered on *one* function, 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 discussion in Section 4). For this reason, selecting a few groups on a call graph results in a data dependence view where only these groups are shown while all others are hidden.

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

4. Dynamic data dependences for program maintenance

In this section, we report on different programming activities around the Blocks World program where we used dynamic data dependences and we discuss our findings.

4.1. 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 acquainted with a program s/he didn't implement her/himself. Two of the authors were in this situation with the Blocks World program and had to work hard to understand the program. 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!

The first view we used for this is the data dependence graph which focused on the called function. Fig. 5 shows this view for the call (creObjet 'a 'boite 'taille '(2 2 3)) that asks for the creation of an object, named 'a', that is a box – *boite* in french – of size 2x2x3. The view shows the input/output of the call, highly uninformative, since the result of the call is just printing out 'c'est fait' (or 'done') and that doesn't say anything about how the program did this. However, this view also shows the global variables (filled in dark gray) that were used and/or modified by the call, information not easily accessible in the interpreter itself. Here we

see that the table before the call was empty³, as was the object list (variables on the top), while after the call, it has been filled with 'a' that also appears in the object list and has properties (variables below the call). With this view, we could discover the real effect of the call.

To better understand how the program functions, we used the user-call-graph, as it gives a first global overview of the actions performed by the program. Of course, this relies on the fact that our program is well decomposed into well named functions: looking at the user-call-graph given in Fig. 4, one can easily grasp that creating an object means first to define the object (defineObject) – this function is recursively called once –, then to find a place where to put it (trouvePlaceSur) and then to actually put this object at this place (metSur); this last action is again decomposed into three steps, namely grasping the object (enleveObjet), closing it whenever it is a box (ferme) and putting it down on the table (deposer).

We then got back to the data dependence graph to get more information about how these different steps affect the global variables. We filtered out everything but user functions and built a first level graph focused on the call to creObjet.

In the case of an object creation, we could verify that the performed actions are always the same. In some other cases, on the contrary, different calls to the same function resulted in really different sets of actions; this was immediately visible in the user call graph and pointed us towards other possible 'traversals' of the program we had to analyze.

4.2. Finding bugs

While working on the data dependence graphs we mentioned in the previous section, we examined in detail how finding a place where to put a object was done. Here, to put an initial 2 by 2 box on an empty table, the program checked whether positions 1-1, 1-2 and 2-1 were free and decided that this was a good place where to put the box. Check of 2-2 was not performed and this didn't produce any error since the table was empty, but of course this also showed a buggy behavior that caused errors in other cases where many objects were already on the table. Incorrectly nested loops were responsible for this error that could be quickly corrected.

Detecting this bug would have been very difficult looking only at the input/output of the program on the Lisp terminal, even when using the built-in inspecting features, while it was straightforward with our data dependence graphs. Fig. 6 shows the corresponding graph, where only three calls, instead of four, to quoiA? are performed, and the argument values indicate which positions were checked.

³nil stands for empty in Lisp.

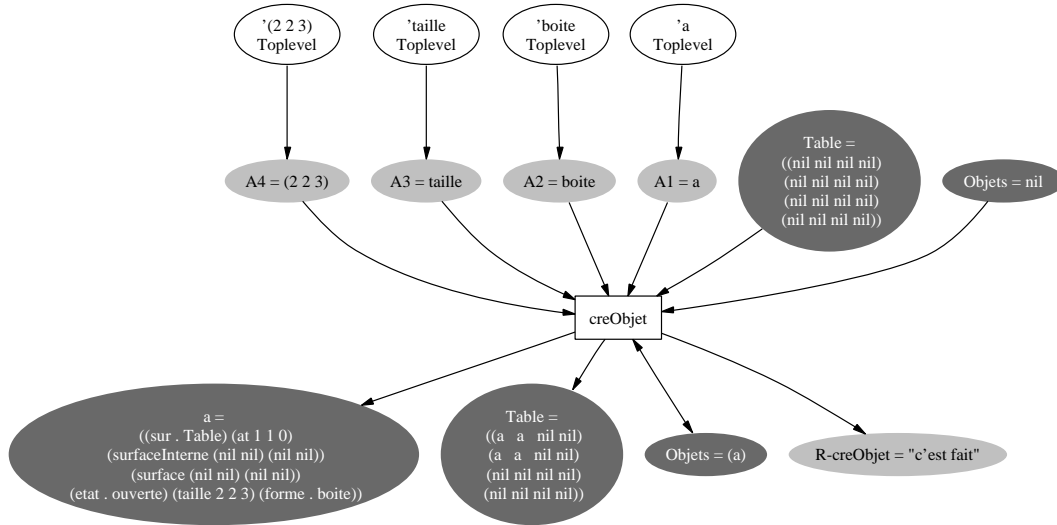


Figure 5. Overview of computation performed

In other cases, we noticed unexpected behaviors of the program and used different possible views to find why it was behaving this way. Our main strategy was to examine the values of the global variables, in iteratively refined views on the different actions performed by the program, to point to a function call working incorrectly. Then we navigated backward and forward to see whether this call was receiving a bad argument – in this case, we again refined views to see what happened *before* this call – or whether it was effectively performing incorrectly. As soon as we had detected the buggy function, we could analyze in more depth what was performed during its call to find the problem. For example, in one such case, where an object had to be moved but was actually not moved, we could detect that a generic sorting function was called with an incorrect function pointer as argument, resulting in not sorting at all. Simply modifying the call solved the problem!

4.3. 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, after correction of the bug in the ‘finding a place’ action, we built several views of calls where this actions was performed and carefully verified that it was, now, correctly implemented.

We also used our views to verify that the program was behaving the way we expected it to do. Remind that it is an AI program, relying on the key concept that most general problems can be recursively solved through a divide and conquer method. That’s why, in many contexts, large parts of the program are reused and reused again, resulting

in deep and broad call trees, extremely difficult to capture.

For example, the user instruction *pose-sur*, that is put-on in english, intended to let the robot move objects in the world, is reused whenever objects are on the object to move – the robot must first *put* these objects *on* the table –, reused (again) whenever the necessary place on, say, the table is not available – the robot must first *put* other objects *on* another place, and then finally it can *put* the initial object *on* some place on the table. This way, a single call to function *pose-sur* may result in it being recursively called several times, each one driving calls to a huge number of other functions each one possibly including non-local returns.

In order to check that this process was correctly implemented, we looked both at user call graphs to check whether function *pose-sur* was recursively called the correct number of times and at a data dependence graph where we rendered visible only calls to function *pose-sur*. With such a view, we were able to examine the values of the global variables at the different steps of the program execution in order to verify that they were modified the way we expected.

5. Discussion

From our experience working with the Blocks World program, as well as several other small to medium sized Lisp 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: details about how execution was driven from one expression to another, as well as about which values variables had at any point of the program and how these values are transmitted from point to point.

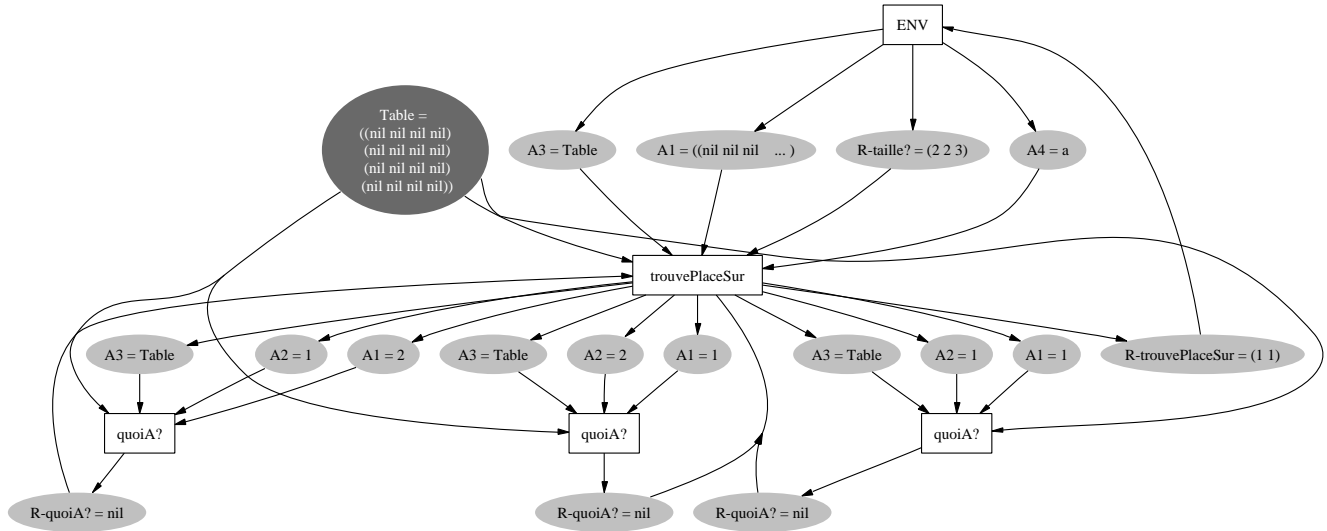


Figure 6. A buggy search for place

The different visualizations we propose were designed to minimize the conceptual overload in order to allow users to find the exact information they need, otherwise barely accessible in the database. Different variants of call graphs respond to questions about the control of the program, and data dependence graphs about the data flow. Clearly, this 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 unforeseen execution paths, sometimes impossible to detect through static analysis. Combining static information with dynamic dependences is a possible extension we plan to investigate.

The second problem we encountered with our tool is that even if the set of dependences is restricted to one execution of interest, it's still sometimes hard to find the right information: either too many nodes and groups are displayed at the same time, or too much navigation is required in the graphs before one finds the place to examine more in depth. For this, we plan to enhance our filtering mechanism with the ability to filter out global variables, since they are not all of interest at the same time, and to implement a query language that will permit to find, thus to jump to, parts of the execution corresponding to given criteria.

Besides enhancements of our visualizations we just men-

tioned, our main perspective is now to development further a similar tool for the C language [4], where we will be able to integrate it with a debugger. This way, the user not only will examine the dynamic dependence graph *after* the execution of the program, but s/he will have the possibility to execute the program step by step, or from breakpoint to breakpoint, while looking at the corresponding graph. We expect this functionality to greatly enhance maintenance of long to execute and hard to understand programs.

References

- [1] T. Ball. The concept of dynamic analysis. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, Toulouse (France), 1999.
- [2] 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.
- [3] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal on Software Maintenance and Evolution: Research and Practice*, 16(3):151 – 185, May/June 2004.
- [4] R. Chaabane. *Analyse Dynamique de Programmes C*. Mémoire de DEA, Université Paris 8, Saint-Denis, France, 2005.
- [5] E. Koutsofios and S. North. *Drawing graphs with dot*. AT&T Labs – Research, Murray Hill, NJ, March 1999.
- [6] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.