Using Dependence Graphs as a Support to Document Programs

Françoise Balmas
Dept. Informatique
Université Paris 8 (France)
fb@ai.univ-paris8.fr

Abstract

In this paper, we argue that source code is not the best support for commenting programs and we introduce dependence graphs as a better support since they offer different levels of abstraction and make visible objects that are not directly represented in the source code.

We first introduce our tool to display dependence graphs, focusing on how different views of the same program can be built and tuned by grouping together nodes that are syntactically or conceptually related. We also show how these views make visible objects of the program that aren't accessible for comment in the source code while they can be, in our tool, richly documented through typed annotations. Then we review which kinds of units can be commented this way and discuss a typical way to apply our tool to document programs. Finally, we give strategies towards the automation of program documentation.

1. Introduction

Program understanding is mainly a process of knowledge recovery. While working with a program, one builds mental representations that explain what the program does or how it works, and this aquired knowledge is likely to be stored: this way one can reuse it later or even share it with other people working on the same program.

The crucial question is thus where and how to store this knowledge. When it concerns low-level units of the program, like instructions or syntactic constructs, it can be attached to the source code, among other existing comments. But when it concerns units that don't have 'reality' in the source file, like the value of a variable after a given loop or a functional module spread over several procedures or even files, it is very hard to efficiently put it directly in the source code, because one doesn't know the precise place where to attach it. This shows that source code, even if it is our primary source of information, is not the best support for comments documenting programs: first it offers only one, very low, level of abstraction – and many comments refer to higher levels like functional or architectural units – and secondly, many objects of the program don't have their counter-part in the source code – since they exist only as the effect of several instructions spread through the source code files –.

In this paper, we explore the role of dependence graphs [1] as support to efficiently document programs at different levels of abstraction. We recently developed a system to display dependence graphs [2] that allows to group sets of nodes [2] in order to reduce the size of the graph. In a fully automatic way, all nodes of structures like functions, loops or conditional statements, are first grouped together, and this recursively in order to form a hierarchy of groups that correspond to the syntactic structure of the program. This hierarchy can then be refined in order to more closely reflect conceptual units of the given program. Dependences – both control and data dependences – are finally displayed with these groups either collapsed or expanded: when most groups are collapsed, the graph is small and shows an overview of the dependences; when groups are expanded to show more nodes, more details are visible but the graph becomes larger; when focusing on a group, the details are visible but the graph remains of manageable size.

With this tool, we are able to see a program at different levels of abstraction: at a low level when we see nodes and almost no group, and at a higher level when no node is visible but only groups; at a very high level, we see only the main components of a program. Furthermore, the corresponding graphs render visible those objects of the program that are not objects of the source code, like the value returned by a function or a variable after a loop.

Therefore, we hypothesized that dependence graphs, organized in such a hierarchical way, are a good support to document programs and we extended our system for this purpose: every group or node, be it an instruction or a variable, can be documented with as many as necessary typed

[1] We use a simplified version of system dependence graphs [8] (see Section 2.1).

[2] A node represents an instruction or a variable (see Section 2.1).

annotations – we provide an annotation editor – and their display can be controlled in order to not overload the corresponding graph. As a result, programs can be documented at different levels of abstraction and objects manipulated by the program can not only be efficiently described but also illustrated with sample values.

In the next Section, we first introduce our dependence graphs, their hierarchical organization and the annotations they are susceptible to receive. Section 3 discusses how dependence graphs are used to document programs in different ways. Finally Section 4 concludes with the main perspectives of this research.

2. Annotating dependence graphs

In this section, we present our system to manage and display dependence graphs: we first introduce our version of dependence graphs; then we show how hierarchies of groups are built and how this allows to display dependence graphs at different levels of abstraction; we then discuss how these hierarchies can be refined to represent conceptual units, this way making visible objects that are not directly represented in the source code; finally we discuss how these units as well as nodes or variables can be annotated.

2.1. Dependence graphs

Several formalisms have been used to represent dependences, like: *program dependence graphs* [10] that include control and data dependences for a single procedure; *system dependence graphs* [8] that are program dependence graphs extended to handle function calls, arguments and returned values; *value dependence graphs* [15] that represent computations of a procedure solely as value flow, with additional representations to handle control information. The dependence graphs we use are a simplified version of system dependence graphs we tuned for our project; we refer to them simply as *dependence graphs*.

A dependence graph contains information both on data and control dependences and integrates all the functions of the corresponding program[3]. Control dependence links connect tests of loops or conditional statements and the statements occuring in the corresponding bodies. Data dependence links connect statements with variables they update and then these variables with every statement where they are used. Such a kind of connection is also used when the variable is updated and used in different functions, even when transmitted as a parameter[4]. Both kinds of dependences are computed by the front-end of the system Canto [1] that integrates a powerful algorithm of pointer analysis ensuring the accuracy of the derived dependences. The corresponding graph is then displayed using the Tcldot package [9].

```
#1      int absolute_val(int x);
#2      void print_result(int nb, float aver, int *tab);

#4      main() {
#5      int i, nb_neg, tot_neg;
#6      float aver_neg;

#8      int tab_numbers[15] = { 12, -5, 7, 3, -89, -45, -6, 9, 10, -1, 0 };
#9      int tab_abs[15];

#11             i = 0;
#12             tot_neg = 0;
#13             nb_neg = 0;
#14             while (tab_numbers[i] != 0) {
#15                     tab_abs[i] = absolute_val(tab_numbers[i]);
#16                     if (tab_numbers[i] < 0) {
#17                             tot_neg += tab_numbers[i];
#18                             nb_neg ++;
#19                             }
#20                     i++;
#21                     }
#22             tab_abs[i] = 0;

#24             aver_neg = (float)tot_neg / (float)nb_neg;

#26             print_result(nb_neg, aver_neg, tab_abs);

#28     }

#30     int absolute_val (int x) {

#32             if (x > 0)
#33                     return(x);
#34             return(-x);
#35     }

#37     void print_result(int nb, float aver, int *tab) {
#38     int i;

#40             printf ("%d numbers of tab_numbers are negative\n", nb);
#41             printf ("Average value is %f\n", aver);

#43             i = 0;
#44             while (tab[i] != 0) {
#45                     printf ("%d\n", tab[i]);
#46                     i++;
#47                     }
#48     }
```

Figure 1. Source code of a sample program

Now let us take a small example program, designed for this presentation, given in Fig. 1: it takes as initial value an array containing positive and negative numbers, puts their absolute values in another array and computes the average of the negative numbers; at the end, both the new array and the average are printed. The dependence graph, in the way we represent it, is shown in Fig. 2.

Every statement is labeled in this example with a number that corresponds to its position in the source file [5] and represented in a black ellipse while variables are in grey ellipses. Data dependences are drawn with two black arrows each, one from the statement where the variable is set or modified to the variable and the second one from this variable to the statement where it is used. For example, towards the middle of the right of the graph, there is an arrow from node #24 to variable aver_neg because this variable is set on line 24 of the program, and then an arrow from aver_neg to node #41 because in this line the value of variable aver_neg is used.

[3] Currently, we don't handle recursive functions.

[4] Actually, we don't represent entry point and parameters of functions, these ones are treated as if they were inlined.

[5] They are usualy numbered according to their position in the abstract syntax tree.

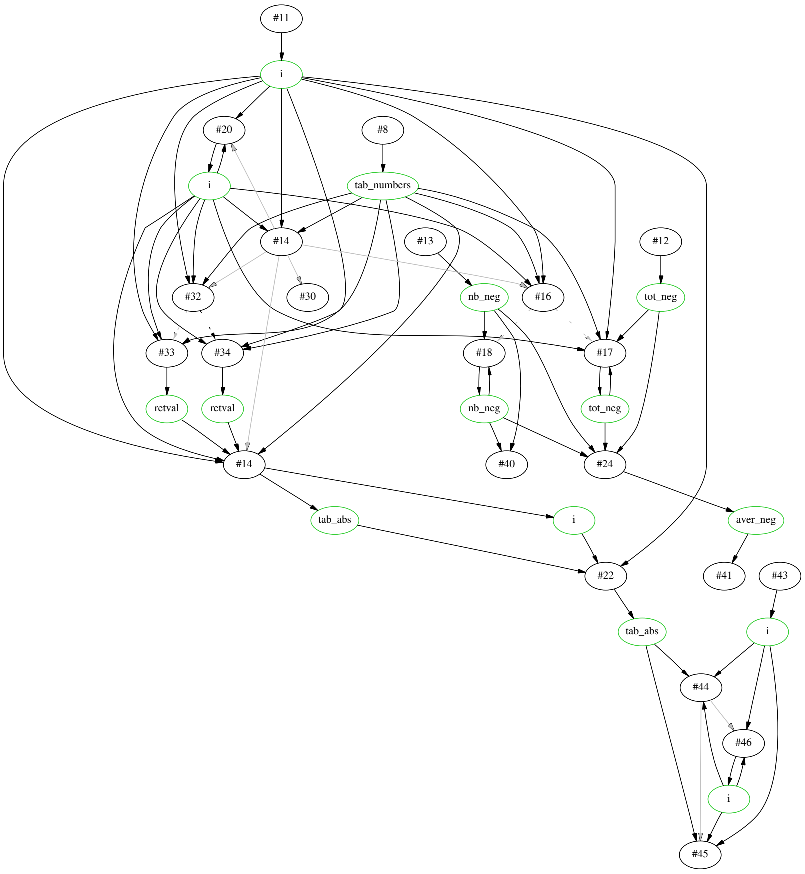

Figure 2. Dependence graph of the sample program

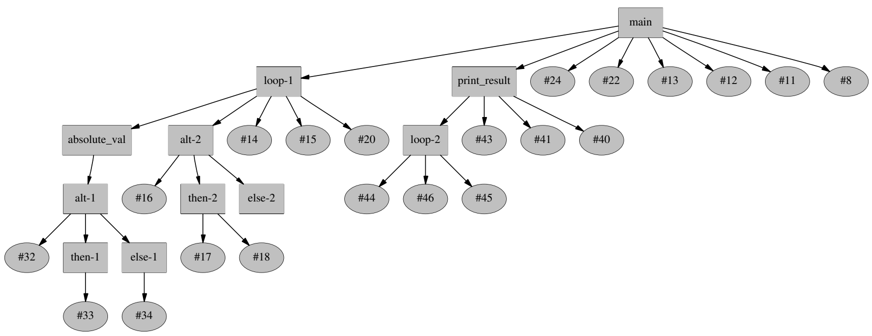

Figure 3. Resulting hierarchy

Control dependences are drawn with a single arrow from the test of a loop or conditional statement to any statement controled by this test; the appearence of the arrow depends on the kind of control: it is grey for the control in a loop, grey dotted for the control in the then branch of a conditional statement and black dotted in the corresponding else branch. For example, in the lower right corner of the graph, node #44 is the test of the loop in function print_result and controls both nodes #45 and #46; on the contrary node #32 is the test of the if in function absolute_val, node #33 is in the then branch and node #34 in the else branch.

This example clearly shows that even for a very small program a dependence graph is complex; the next two sections introduce our mechanism to render these graphs more readable.

2.2. Structural hierarchies

In our tool, nodes are automatically grouped according to structural criteria: nodes belonging to a loop are grouped together, as well as nodes belonging to the then or the else branch of a conditional statement, and nodes belonging to the same function definition [6]. If a loop contains a function call, the group representing this function is grouped with the other nodes of the loop; in the same manner, a loop appearing inside the then of a conditional statement is represented by a group that in turn is grouped with the other node of this branch to form a single group representing the then clause.

Let us take again the example given in Fig. 1. The groups automatically built for this program are visible in the hierarchy shown in Fig. 3 where boxes represent groups and ellipses nodes (variables are not visible in this representation). All statements of the loop of function print_result have been grouped under the group loop-2, then this group has been grouped with the other statements of the function under the group labeled print_result which is in turn grouped with loop-1 and the remaining statement of function main. Similar groupings have been done to represent the loop of function main, labeled loop-1: it contains the call to function absolute_val and a conditional statement, labeled alt-1 and containing the test (node #16), the then and the else branches.

The construction of the hierarchy is done once when loading a new program and then one can use it to choose how to display the dependence graph. Suppose we first display in our GUI the graph shown in Fig. 2. By shift-clicking on say node #44, in the lower right corner of the graph, we ask to *collapse* this node together with the other nodes of group loop-2 (see Fig. 3). The net result of this action is that only loop-2 remains visible – and drawn as a box, at the bottom of the graph – while all its content is hidden (see Fig. 4): the dependences between #44, #45 and #46 are no more visible while dependences between one of these nodes and an other node – for example from #43 to #46 – became dependences between loop-2 and the other node – from #43 to loop-2 in our example–. We can continue this process by collapsing the groups that are under loop-1, beginning at the bottom and stopping when loop-1 is collapsed [7]. The corresponding graph (Fig. 5), shows the program at a higher level of abstraction: it shows that it basically proceeds in two steps, first computing a set of values (loop-1) that are then used in the second step (loop-2).

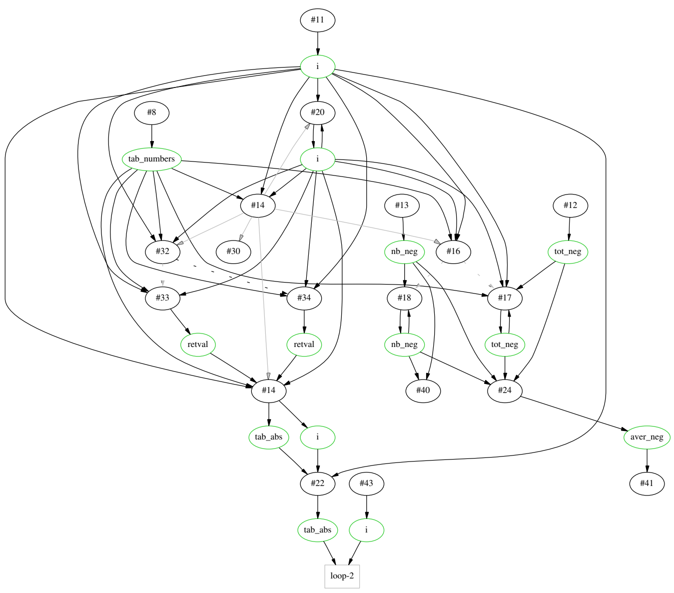

Figure 4. The dependence graph with one group collapsed

At this point, one can either continue to collapse groups, or, inversely *expand* already collapsed groups. Ctrl-clicking on a displayed group asks for the group to be replaced by all the nodes or groups belonging to it, making the graph larger but more detailed. Alternatively, one can *focus* on a group by alt-clicking on it: this way only the content of this group is visible, all other nodes or groups being hidden. For example, Fig. 6 focuses on group loop-2, showing only nodes #44, #45 and #46 and their dependences. Every dependence for which the source is outside loop-2 is drawn as coming from the virtual group <BEFORE>loop-2 – here the initial value of i and the value of tab_abs –; in the same manner, every dependence for which the target would be outside loop-2 would be drawn as going to <AFTER>loop-2.

[6] Actually, one may also select only one or two criteria to build the groups [3].

[7] Note that this process is handled automatically by our tool.

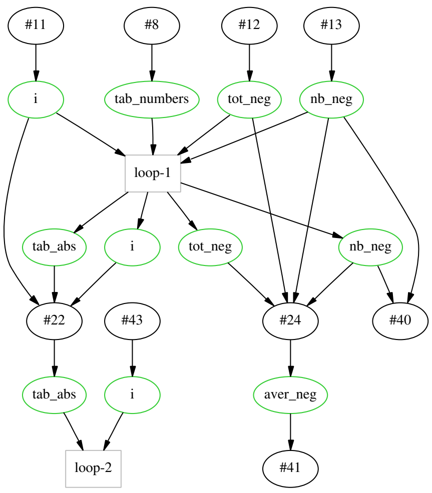

Figure 5. Almost all groups collapsed

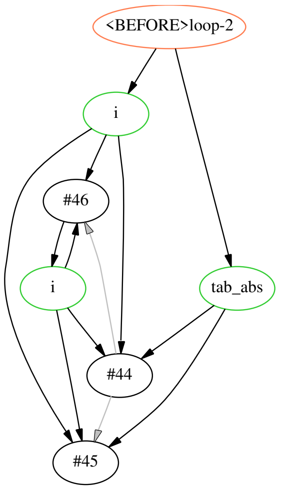

Figure 6. Focus on the content of a group

2.3. Conceptual hierarchies

By allowing to browse dependence graphs according to an automatically built hierarchy of groups, our tool provides an efficient way to analyze programs at different levels of abstraction. However, the corresponding groups are not always satisfactory: we would like for example to have the initialization of loop variables grouped with the loop itself, or computations split in several instructions represented as a single group. These are concepts that don't exactly correspond to a single syntactic structure.

Our GUI provides a way to manually build groups exactly for this purpose. At any moment, one can group together selected nodes and this way refine the existing hierarchy. For example, in Fig. 5, we can logically group together node #43 with group loop-2 as it is just an initialization for the loop. For the same reason, we group nodes #11, #12 and #13 with loop-1, and we also group with these ones nodes#22 and #24 as they terminate the computations performed during the loop. Finally, we group together nodes #40 and #41 since they both have the same purpose: printing information about the negative numbers and their average. The resulting refined hierarchy is shown in Fig. 8 (where the new groups have been drawn in white instead of grey). The resulting dependence graph is highly simplified (see Fig. 7 to be compared to Fig. 5) and clearly shows the structure of the program while hiding all the implementation details: there is one first step where values are computed and then two different values are printed. This is easy to catch because we have only three groups visible, even when each one corresponds to several instructions spread through the source code.

Note also that grouping together nodes or groups that produce values for the same variable – like in in this example node #12 and group loop-1 that update variable tot_neg – ends up with a single group updating a variable; this last object of the dependence graph represents the final value of the variable after a given computation that doesn't correspond to a single instruction in the source code, but results from the same kind of fusion of different statements as performed in the Single Static Assignment [6] technique.

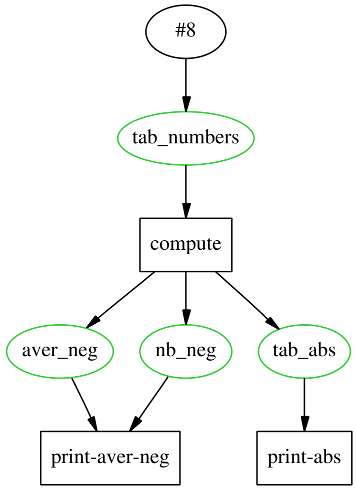

Figure 7. Dependence graph with conceptual computations

In conclusion, with the basic existing hierarchy and its possible refinement, we have the ability to represent as an object of a dependence graph any unit of computation and any value of a program. This way, we can easily comment them, as we do for other items, with the annotation mechanism presented in the next section.

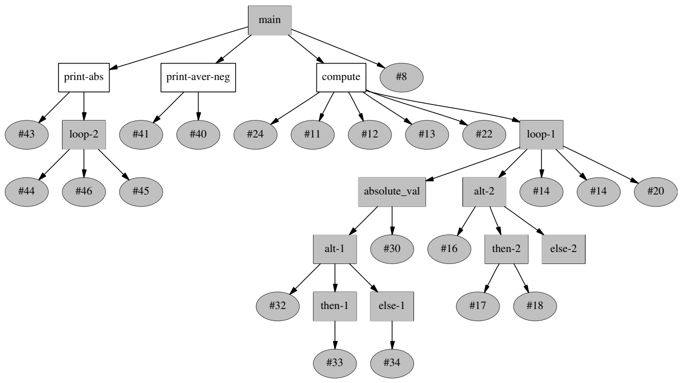

Figure 8. The modified hierarchy

2.4. Typed annotations

Any object of a dependence graph, be it a node, a group or a variable, can receive any number of *typed annotations*, namely a piece of text, the *annotation* itself, linked to a keyword, the *type*. Such typed annotations can be used to document a function with its number of arguments, their types and its return type, or a variable with its role, its type and its range of value, for example.

Typed annotations are attached to the objects of the dependence graphs and can then be modified at any time. They can be visualized on demand – simply by pressing a mouse button – or displayed inside the graph itself: when a typed annotation is displayed, its content appears under the name of the object itself.

For example, in Fig. 9 we put annotations on the groups and variables of Fig. 7. The comm annotation shortly describes the effect of each step of the program and the role of each variable while the type annotation gives the type of each variable.

As this example shows, the annotations allow to have more information than what is easily extracted from the dependences. However, displaying all the typed annotations is rarely suitable, as this greatly overloads the graph. That is why we control the display of typed annotations for each object in the following way:

- a typed annotation can be either always, or on the contrary never, displayed (for example, the comm annotation of every object that has this annotation is always/never displayed)
- each typed annotation of every object integrates a flag that states if it is to be displayed or not; this flag is used whenever the global status of a typed anotation is unknown.

This way, we can tune views that show the exact amount of information we choose to document given units of programs.

3. Documenting Programs

In this section, we first review the different kinds of units we can annotate using our tool, we then present a typical way to apply it to document programs and we compare our tool with other redocumentation tools.

3.1. Units

Due to its capabilities to group together several related nodes or groups, and in this way to present as a single object a variable which may have been set in different statements, our tool allows to document different units of programs that are not easily accessible in the source code. We review them in the (open) list below.

Structural units (functions, loops or conditional statements) Nodes are grouped on the basis of syntactic structures, that are usually easily identified in the source file; however, since such structures may spread over pages and pages of code, representing them as a single group allows first to show a global outline of their behavior and secondly to clearly identify the scope of eventual comments.

Conceptual units As Section 2.3 showed, conceptual groups allow to put together nodes or groups that work towards a given computation. These may be the initializations or terminations done around a loop, or a sequence of statements that progressively produce a final value; in these cases, a single group represents the computation even if it may be hard to identify the corresponding statements in the source code.

Note that we created such groups when we built the graph of Fig. 7.

Threads of computations Often computations split into several statements are mingled even if they are actually not directly related – say, they use shared values but produce different values –. Inserting comments in such sequences of lines of code is not straightforward, because comments also are mingled and thus hard to follow. Such sequences of statements are represented in the correponding dependence graph as separate chains of nodes, that can thus be separately annotated and separately grouped. This shows that our approach accounts naturally for delocalized plans [12].

For example, in Fig. 10 the computation of a point coordinates, x3 and y3, is done mingling two sequences of statements. Commenting separately these sequences is a bit complex if one wants to globally comment each computation. On the contrary, in the dependence graph, one can collapse each computation

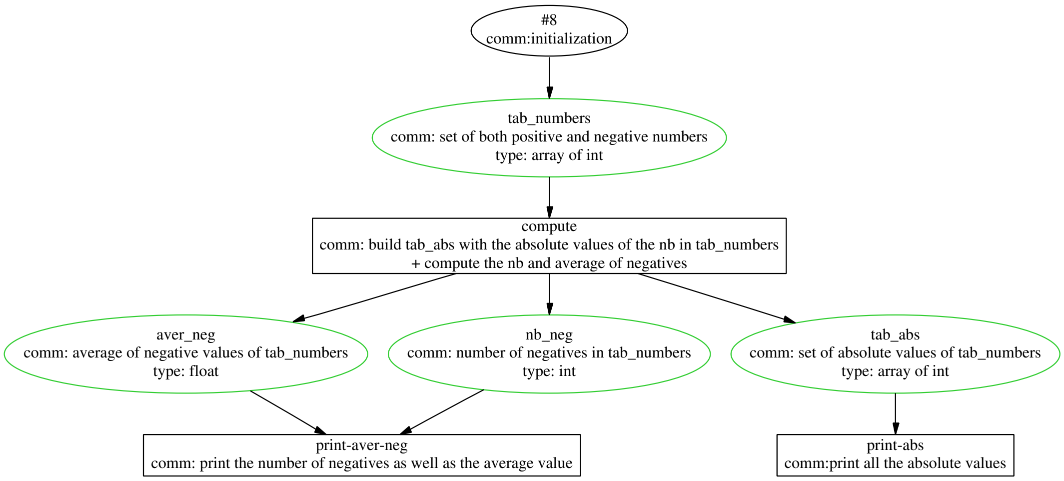

Figure 9. Dependence graph with typed annotations

```
a = x1 + y1;
b = x2 + y2;
c = a * 80;
d = b * 120;
x3 = c + 5;
y3 = d + 15;
display(x3, y3);
```

Figure 10. Mingled computations

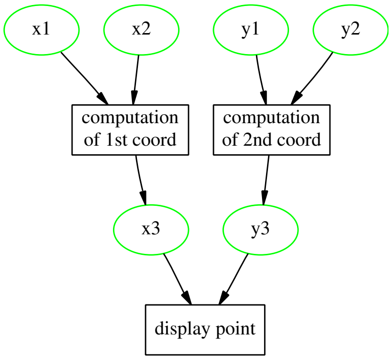

Figure 11. Separate comments

in a conceptual group and comment them separately, as Fig. 11 shows.

Such a representation is also useful when, on the contrary, values are shared by different computations. If, for example, we change the computation of variable d in Fig. 10 to

```
d = b * 120 + a
```

we get the annotated dependence graph of Fig. 12 that clearly shows that x1 and y1 are used to compute both coordinates.

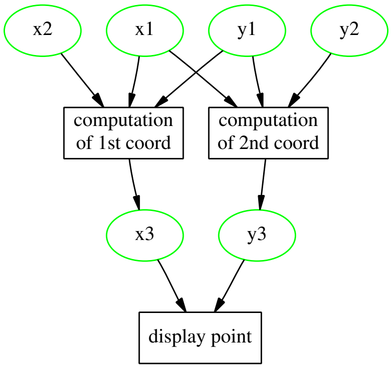

Figure 12. Shared computations

Variables resulting from groups As already pointed out in Section 2.3, when different statements producing a value for the same variable are grouped together, this leads to a single group producing a single variable. This means that we have in the dependence graph a single object that represents the final value of the variable, after a unit of computation: a function call, a loop, a conditional statement or a more complex combination of such. In all cases, the point in the source code where a comment on this final value should be inserted – at the end of the loop, just after the function call, etc. – is usually where the variable is neither modified nor used, thus in a point where nobody would look for such a comment. In the corresponding dependence graph, comments are attached to the variable resulting from the group and are accessible either when one analyzes the group itself or the possible different usages of the variable thereafter.

In Fig. 13, for example, a loop computes the sum of the elements of an array. This sum is the result of an initialization before the loop followed by an iterative increment during this loop. Its final value, which is actually the sum of *all* the elements of the array is accessible only after the loop and that's where a comment on this fact should be inserted. The drawback is that at this point there is no statement containing the variable, thus no reason to attach such a comment. One would more probably look for it later when the variable is used, but if this variable is used several times, the comment is not likely to be duplicated. In the dependence graph (see Fig. 14), on the contrary, the comment is attached to the variable resulting from the group that represents the whole loop (initializations and iteration) and then is 'transmitted' to every point where the variable is used: even if one of these points is several pages later in the file or in another file, it appears in the graph close to the place where the variable is set or used.

```
x = 0;
i=0;
while (i<N) {
        x += tab[i];
        i++
        }

/* comment on final value of x should come here */

...

/* comment looked for here, when x is used */
func1(x);

/* or here, when x is again used */
func2(x);
```

Figure 13. Commenting the 'result' of a loop

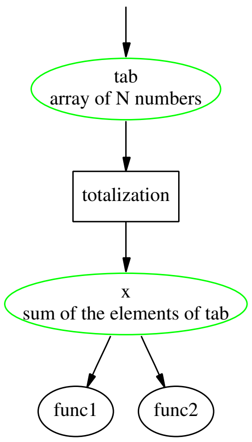

Figure 14. Annotating the 'result' of a loop

Note that one of the perspectives opened by this research is to build a catalog of those units of programs that are not directly represented in the source code and to verify to which extent they are accessible in dependence graphs.

3.2. A typical application

We used our tool to document several programs, in particular in the context of a seminar [17]: our goal was to build documentation views that would help students understanding what programs computed and how they worked. We found that typically three classes of comments were helpful. We discuss them below using, for sake of simplicity, the program of Fig. 1 as example.

Comments at code level The purpose of the first class of comments is to bridge the gap between the program itself and its representation as dependence graph. Actually, identifying to which computation a set of nodes correspond is very hard because there is no information in nodes but line numbers. That's why we put annotations on each node with the corresponding code and a short sentence describing its purpose; we also put annotations on each variable in order to comment its role, its type and whenever possible its actual value.

Fig. 15 shows an example of such annotations attached to the nodes and variables of Fig. 6.

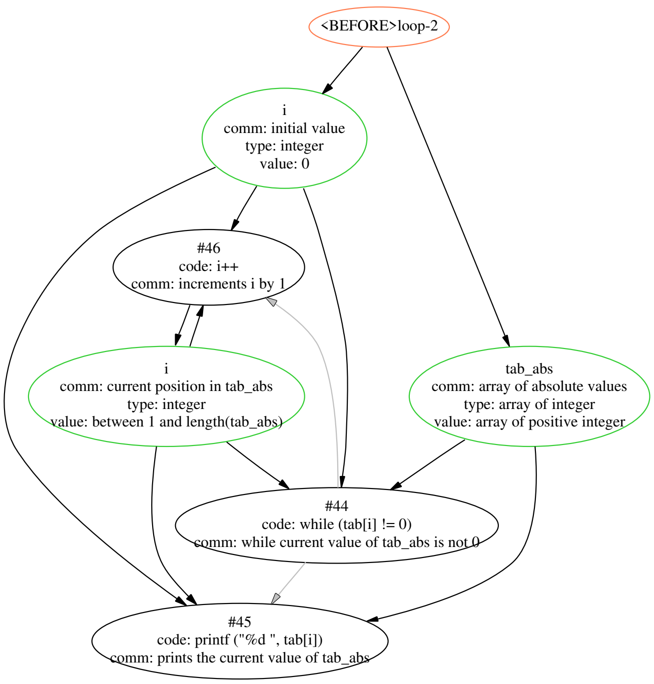

Figure 15. Comments of the code

Comments at conceptual levels The purpose of the second class is to comment those units of programs we mentioned in Section 3.1. We basically inserted annotations to explain the global effect of structural or conceptual groups and detailed the roles and possible values of the variables resulting from these computations. Fig. 9 shows examples of such annotations.

Sampling variables While preparing documentation views for our class, we noticed that documenting the role and possible values of variables was not always sufficient to explain given programs, since actual examples were missing. That's why we added annotations that show the real values taken by the different variables for a given input or initial value. For example, Fig. 16 shows the successive values (one below the other) taken by variables i during the iteration over the elements of tab_abs.

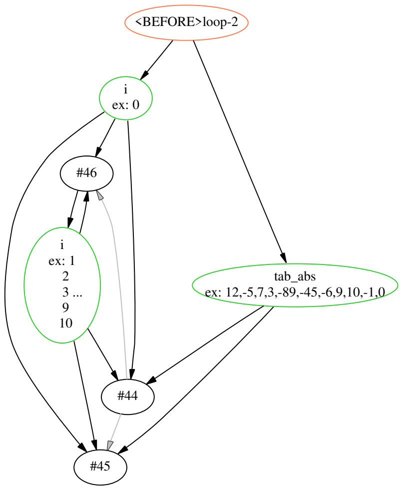

Figure 16. Examples of variable values

Note that we generated these annotations automatically after manual modification of the programs: special directives were inserted at given points we chose carefully in order to get all the necessary values for the views we wanted to build. This way, we can automatically generate the corresponding annotations for any possible input or initial value and then display them on demand, altogether or on the contrary one after another in order to create an animation of the program susceptible to assist its comprehension [4]. Automating the instrumentation of programs is of course one of our main perspectives.

Redocumentation [5] is the creation of semantically equivalent representations of software, is a central activity in Reverse Engineering and Software Maintenance. Several tools have thus been developed to help people create, store and browse documentation views of programs. In [16], a programming environment is described that automatically integrates comments found in those source files that are loaded and that proposes a specialized editor to attach different kinds of annotations to any point of the corresponding programs; as a result, programs are documented at different levels of abstraction and different view points can be displayed. In RIGI [18] graphs representing artifacts of the program under consideration can be manipulated in order to build representations at different levels of abstraction; annotations can then be attached to these representations. In [14], authors propose an approach to build different level of abstraction documentation, combining automatic extraction from source code and hand-written information; this process can easily be tailored to specific reverse engineering needs.

The specificity of our tool is to provide a way to naturally document units of programs that are not directly accessible in the source code. This enables a uniform representation – dependence graphs – to be the basis of different levels of abstraction views that always remain closely related to the program itself; this ensures consistency between documentation and source code. Furthermore, the use of typed annotations allows for a wide variety of comments that can be tailored to specific tasks or to different users' levels of expertise [7], as well as for incremental redocumentation of programs [11], since annotations can be edited or extended at any moment.

4. Conclusion

In this paper, we argued that source code is not always the best support for comments, since it offers only a low level approach to programs and since many units are not directly accessible, either because the corresponding code is spread over several pages or because they do not correspond to one single point in the source file.

We then showed that dependence graphs are a better documentation support. Our tool allows to automatically group syntactically related nodes together and then to refine these preliminary groupings by collapsing conceptually related nodes or groups. This way, we can tune graphs that show units of computations and the values they produce: all these units are represented by objects of the dependence graphs that can be documented through typed annotations.

An annotation editor, added to a dependence graph browser, was used to document several programs in the context of a class. For this purpose, we worked with programs of rather small size and we could attach all the necessary annotations manually. Would these programs have been of real-world size, it would have been too much an effort to be affordable.

This leads to the question of the scalability of our approach. In [3], we showed that the hierarchy construction scales well to programs of larger size, producing groups with a satisfying average number of nodes: in most cases, the number of nodes in a given group is not greater than twenty; furthermore, this value, seems clearly to depend much more on the syntactic structure of the program than on its size. Scalability for documenting means primarily automating documentation, and this clearly is not so much an algorithmic question but a conceptual one. However, strategies can be envisaged to automate the construction of (part of) a dependence graphs documentation:

- the code corresponding to nodes can be extracted by a parser, modified in order to not only build the abstract syntax tree, but also to memorize each instruction; such a parser could also extract information on types;
- the comments already existing in the source files can also be extracted and stored as annotations; the main problem is to identify to which unit of computation a comment really refers; this is a question we are currently investigating;
- actual values of variables are accessible using a debugger like `gdb` [13]; here the question is to decide where to put breakpoints in order to collect the right set of values; automating this, clearly, requires further research.

Our plan is to develop a tool able to extract as much as possible from source files and during execution of programs in order to build a documentation basis, susceptible to be augmented and tuned, using our dependence graph manipulation tool, during reverse engineering activities: our conviction is that such a system will be a good basis to store, browse and share documentation of large scale software.

References

[1] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the CANTO environment. In *Proceedings of the Int. Conf. on Software Maintenance*. IEEE Computer Society Press, 1997.

[2] F. Balmas. Displaying dependence graphs: a hierarchical approach. In *Proceedings of the Workshop on Analysis, Slicing and Transformation, WCRE 2001*, Stuttgart, Germany, 2001.

[3] F. Balmas. A hierarchical approach to display dependence graphs: tool and evaluation. Technical Report TR-2002-3, Université Paris 8, Saint-Denis, 2002.

[4] E. Burd, D. Overy, and A. Wheetman. Evaluating the use of animation to improve the understandibility of sequence diagrams. In *Proc of the 10th Int. Workshop on Program Comprehension*, Paris, France, 2002.

[5] E. Chikofsky and J. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1), 1990.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proc. of the 16th Annual ACM Symposium on Principles of Programming Languages*, 1989.

[7] A. Erdem and W. Johnson. Task orientation and tailoring of interactive software explanations. In *Proc of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, 1999.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, January 1990.

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

[10] K. Ottenstein and L. Ottenstein. The program dependence grpah in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, 1984.

[11] V. Rajlich. Incremental redocumentation with hypertext. In *Proc. of Eutomicro '97*, Berlin, Germany, 1997.

[12] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.

[13] R. Stallman, R. Pesch, S. Shebs, and al. *Debugging with GDB – The GNU Source-Level Debugger*. Free Software Foundation Inc., 2002. 9th edition for GDB version 5.1.1.

[14] A. van Deursen and T. Kuipers. Building documentation generators. In *Proc. of the Int. Conference on Software Maintenance*, Oxford, England, 1999.

[15] D. Weise, R. F. Crew, M. D. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the Twenty First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[16] H. Wertz. The design of an integrated, interactive and incremental programming environment. In *Proc. of the 6th Int. Conference on Software Engineering*, Tokyo (Japan), 1982.

[17] H. Wertz and F. Balmas. Cours de dea, option 'environnements de programmation'. Université Paris 8, Departement Informatique, 2002.

[18] K. Wong, S. Tilley, H. Mueller, and M. Storey. Structural redocumentation: A case study. *IEEE Software*, 11(6), 1995.