Displaying dependence graphs: a hierarchical approach

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

Abstract

We present a method to handle data- and control-flow information, represented as simplified system dependence graphs. As soon as a program is bigger than a few dozens of lines of code, its dependence graph becomes unreadable with standard drawing tools, since it contains far too many nodes and edges. In our approach, we propose to decompose the program into a hierarchy of groups that are likely to be of manageable size. We implemented a tool that first builds this hierarchy and stores it in a database. A graphical interface allows then to browse this hierarchy to visualize the dependences of each group, to annotate the nodes or groups and possibly to refine the proposed hierarchy.

This paper introduces our approach for program decomposition, it describes our tool for dependence exploration and discusses the preliminary results we obtained with a few sample programs.

1. Introduction

Identifying that every piece of a given program is strictly necessary, in other words that a program doesn’t contain any extra computation, not required by the specification, is crucial in different contexts.

In legacy systems, when computations have been modified or replaced by new code, some unnecessary computations may still remain in the code, either because the programmer left it for the case it would again become necessary or because s/he introduced new code without taking into account that the other computation still existed. Detecting these unnecessary computations would allow to clean up the code and make it more understandable and easier to maintain.

When purchasing software, some companies with sensitive information in their networks that they don’t want to be accessed by outsiders, may wonder if this software doesn’t integrate any “spy” modules that could be able to access their data or steal it. If such functionality could be detected, this would allow companies to avoid using unsafe software.

Finding ways to deal with this problem of possible extra computations implies to find efficient ways to explore how the data – mostly handled through variables – of a program are manipulated, in particular where they are defined and then where they are used: variables defined, but not used, or used for an unknown specification, are good candidates for further analysis. One solution is to compute the different flows of the program and to analyze them, for example with slicing tools like CodeSurfer [3], Unravel [10] or Ginsu [9], but finding the right variable to slice may be a very hard task. An alternative solution is to represent the flows, with dependence graphs [4], for example, that show how the variables are used: just looking at such graphs may help to point to the right place, which needs further analysis, possibly with other tools.

The drawback of this solution is that the dependence graph of even a small program contains too many nodes and edges to allow the graph to be easily analyzed. CANTO [1] and Chopshop [6] are attempts to address this problem: CANTO allows to focus on one node with a given number of upward and downward levels; Chopshop performs preliminary abstractions on procedure calls that dramatically reduce the size of the resulting graphs. The motivation of our work is to propose a more flexible solution: according to three possible criteria, programs are decomposed into meaningful chunks for which the dependence graph is likely to be of manageable size. These chunks or groups are hierarchically organized, that hierarchy being possibly refined later. At every level of this hierarchy, the dependence graph can be analyzed and annotated\(^1\).

Our paper is organized as follows. In Section 2, we introduce our decompositions of programs into a hierarchy of groups; Section 3 describes the system we implemented to automate this process and to browse the corresponding dependence graphs; Section 4 discusses our approach while Section 5 proposes directions for further research.

\(^1\)Actually, as explained in Section 3, we use a simplified version of system dependence graphs [5] we simply refer to as dependence graphs.
2. Program decomposition

Drawing the dependence graph of a program, even if the program is small, requires to draw such a large number of nodes and edges that the corresponding graph is unreadable when not impossible to draw at all. ¹

When the graph can be drawn, one solution in order to deal with numerous nodes and edges is to collapse chosen nodes in groups, the effect being that less elements are displayed, a few groups instead of a lot of nodes, and all inner edges – edges connecting grouped nodes – are hidden. Groups can further be grouped with other nodes, forming a hierarchy of groups that finally transforms the graph into a manageable size.

For example, consider the sample program given in Fig.1. Its initial dependence graph is given in Fig. 2: black ellipses represent positions in the program and grey ellipses variables; black edges show the positions in which variables are updated and where the resulting values are used; grey edges show control dependences in the loop. ² If we want to simplify the graph, we can for example group the nodes of the loop together (namely groups #3, #4 et #5), then group the loop with its initializations (nodes #1 and #2) and finally group together the two nodes printing the result (nodes #6 and #7). This grouping process can be represented by the hierarchy given in Fig. 3 where groups are represented as boxes and nodes as ellipses. Fig. 4 shows the resulting dependence graph when all groups have been built: only two data dependences are still visible.

Several systems enable grouping of nodes. For example, RIGI [17] integrates a graph drawing tool that allows to collapse nodes that have been selected, one by one, by the user. The drawback of this method is that deciding which nodes to group together may be tedious if the initial graph is large. On the contrary, dotty [7] and veg [13] allow to group nodes according to predefined criteria. In dotty, nodes with the same value for a given attribute can be grouped. In veg, graphs may include sub-graphs, these sub-graphs being possibly folded. This way, grouping becomes straighforward if

²For example we were unable to draw the dependence graph of a 3000 LOC program with dot on a IRIX 6.5 machine with 128 Mbytes of RAM and 1 Gbytes of disk space available.

¹Details on our representation of dependence graphs are given in Section 3.
the sets of nodes to group are correctly prepared. On the other hand, the list of possible groups is set once and for all and cannot be modified at run time.

Our approach is to automatically build a hierarchy of groups that is stored in a database, to supply an interface through which this hierarchy can be browsed – thus the dependence graph of every group of the hierarchy can be displayed (see Section 3.3) – and to allow for the refinement of the initial hierarchy through updates of the database. We propose three possible decompositions we present in the rest of this Section.

• Decomposition into function levels

In this decomposition, all nodes belonging to a given function are grouped together, the hierarchy of groups being thus isomorphic to the function call graph.⁴

Consider the pseudo-program given in Fig. 5 in which statements have been replaced by node numbers. Fig. 6 shows how these nodes are grouped: for example, nodes n2, n3, n5, n8 and n10 are grouped together because they belong to the same function and the corresponding group, funcA, is grouped with n13 and n14 because it represents the call to function A performed in function C where n13 and n14 are also performed.

• Decomposition into loop levels

In this decomposition, function calls are seen as if the functions were inlined and only the nesting of loops is considered. A node is thus grouped with all the others that are visited during the execution of the same loop.

In Fig. 7, we show this decomposition for the program of Fig. 5. Here we can see that nodes n1 and n11 belong to the same group, loop2, even if n1 is in function B and n11 in function main, because they occur during the same loop.⁵ Note that the boxes with the function names are only given as an help to understand the figure and do not correspond to anything in the corresponding hierarchy.

• Decomposition into function and loop levels

This decomposition is a combination of the last two as both loops and functions are grouped: therefore function groups contain simple nodes, other function groups and loop groups; similarly, loop groups contain simple nodes, other loop groups and function groups. This way, function calls occurring in loops can be clearly identified, since they are represented with a single group. The same property holds for loops occurring in functions.

⁴Calls through function pointers and possible cycles in the call graph are not handled in the current version.

⁵Note that we don’t take into account the different possible paths in a given loop.

\[
\text{func Main} \\
n9 \\
\text{begin loop} \\
n16 \\
n12 \\
n11 \\
call func B \\
\text{end loop} \\
call func C \\
\text{end func Main}
\]

\[
\text{func B} \\
n15 \\
n8 \\
n7 \\
n4 \\
\text{end loop} \\
n1 \\
\text{end func B}
\]

\[
\text{func C} \\
call func A \\
n13 \\
n14 \\
\text{end func C}
\]

\[
\text{func A} \\
n10 \\
n6 \\
n5 \\
n3 \\
n2 \\
\text{end loop} \\
\text{end func A}
\]

**Figure 5. A sample program**

Fig. 8 shows this decomposition for the same program. Here, n1 is grouped in function B while n11 is in loop2.

Note that these three decompositions are based on the control flow of the program: this ensures that we group together nodes that are computationally related. Other decompositions could also be envisaged; we will discuss this further in Section 5.

3. Our system

In this section we present the system we implemented that builds a chosen hierarchy and allows to explore the corresponding dependence graphs as well as to refine the initial hierarchy.

Several formalisms have been used to represent dependences, like: program dependence graphs [11] that include control and data dependences for a single procedure; system dependence graphs [5] that are program dependence graphs extended to handle function calls, arguments and returned values; value dependence graphs [16] 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 sys-
Figure 6. The function level hierarchy

Figure 7. The loop level hierarchy

Figure 8. The function and loop level hierarchy
tem 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. Control dependence links connect tests of loops or ifs and the statements occurring in the corresponding bodies. Data dependence links connect statements which 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 parameter.

For example, in Fig. 2, node #3 is the test of the loop and thus controls the body, nodes #4 and #5 (grey edges); n is updated in statement #1 and its value is then used in three statements, #3, #4 and #5 (black edges). If the compute-fact function of Fig. 1 is called in the following way:

```plaintext
[10] val = 5
```

node #2 and #3 are directly connected to variable val, without any mention to variable x (see Fig. 9).

Our system works in three main steps (see Fig. 10) that are detailed below.

### 3.1. Parsing

The goal of the first step is to extract low-level information from the program. For this, we rely on the system presented in [15] that parses preprocessed C source files in order to build an abstract syntax tree (AST) and to compute the dependences. As this system integrates a powerful pointer analysis algorithm, it provides accurate details on data dependences.

From the output of this system, we dump the data dependences in the following format:

Refers pB variable pA

which means that position B of the program – a node of the AST – uses a given variable that is defined/modified in position A.

We also extract information about the control structure of the program, namely the possible paths in the AST, the entry points of functions or loops as well as the branching of ifs. This information is used to build the hierarchy (see 3.2) and to compute the control dependences we store in the following format:

LDep pB pA
TDep pB pA

---

6Remind that our goal is to visualize dependence graphs, in order to examine variable usage, not to use graphs representations for further analysis.

7Actually, we don’t represent entry point and parameters of functions.

---

Figure 9. The graphical interface

EDep pB pA

which means that position B is controlled by position A because it is inside a loop (LDep) and position A is the test of the loop; or it is in the true, resp. false, part of an if (TDep, resp. EDep) and position A is the test of the if.

The data and control dependences form the dependence graph that is displayed in the third step (see 3.3).

### 3.2. Decomposition

During the second step, the hierarchy of groups is built according to the chosen decomposition, namely function, loop or function-loop decomposition as introduced in Section 2. The corresponding algorithm is given in Fig. 11.

The result of this process is, for each group of the hierarchy, a definition with the nodes and sub-groups it contains. These definitions are stored in the data base we maintain in order to control which nodes and edges have to be displayed. Each entry of the data base contains the following items:

- the name of the node/group
- its type, name or group
- for groups, the list of nodes and sub-groups it contains
1. take main-level as current group;
2. take node 1 as current node;
3. insert current node in the definition of the current group;
4. get next node (following the path through the syntax tree):
   (a) if this node is a simple node, go to step 3;
   (b) if this node is an if, then iterate step 3 and 4 for both then and else branches and then with the node following the if statement;
   (c) if this node is a function call:
      i. if the chosen decomposition is function or function-loop, then create a new group, insert it in the definition of the current group, push the current group and then the new group is promoted the current group;
      ii. iterate step 3 and 4 for the body of the function;
      iii. if the chosen decomposition is loop or function-loop, then pop the current group;
      iv. iterate step 3 and 4 for the node following the function call;
   (d) if this node is the entry point of a loop:
      i. if the chosen decomposition is loop or function-loop, then create a new group, insert it in the definition of the current group, push the current group and then the new group is promoted the current group;
      ii. iterate step 3 and 4 for the body of the loop;
      iii. if the chosen decomposition is loop or function-loop, then pop the current group;
      iv. iterate from step 3 and 4 for the node following the loop statement;
   (e) if no node – end of program reached –, then stop.
5. dump every group definition

Figure 11. Decomposition algorithm

- (optional) a text describing the node/group (allowing for possible annotations)
- a flag stating:
  - in case of a group, if it is open or closed,
  - and in case of a node, if it is visible or hidden.

At this stage, every group is closed, leaving only the main-level group visible.

3.3. Visualizing

A graphical interface is provided to visualize the dependency graph and to browse the different groups built during the previous step.

Shell scripts are run to update the dependences, looking at the status of nodes and groups: for each node of every dependence, we search if it is visible or hidden; if it is hidden, we look for its group, then we search if it is open or closed; if closed, we iterate the process until we find an open group. If both nodes/groups of the resulting dependence are the same, we don’t display it at all; otherwise, we dump (in the dot graph description language) it in the following way:

- for control dependences, we dump both nodes/groups and an edge between the two (directed from the controlling node to the controlled one);
- for data dependences, we dump both nodes/groups as well as the variable, with an edge from the defining node to the variable and a second edge from the variable to the using node;
- nodes are drawn as black ellipses, groups as black boxes and variables as grey ellipses.

These elements are then processed by the dot tool [8] and displayed with a Tcl/Tk program using the Teldot package 8. This way we provide an interface that enables to interact with the displayed items in order to perform actions suited to the browsing of the decomposition hierarchy. Fig. 9 shows the graphical interface of our system.

This interface offers the following actions:

ungroup control-click on a group opens the group by ungrouping all its nodes/sub-groups that become visible; the database is updated accordingly.

8 Teldot is part of the dot distribution.
regroup shift-click on a node/group finds to which group this node/group belongs and close this group, regrouping all its nodes and groups, thus making them hidden; the data base is updated accordingly.

focus alt-click on a group both ungroups this group and regroup all the embedding groups, making thus only this group and the main group visible; in this case, every edge coming from the main group represents an event – variable updating for instance – performed before the current context (function or loop), while an edge going to the main group represents an event – variable using – occurring after this context; the data base is updated accordingly.

select click on a node or group selects it (see below).

group button2-click creates a new group; a new entry is created in the data base, with all the selected nodes and groups as members of the group\(^9\); the new group is closed. This interaction enables the refinement of the initial hierarchy.

\(^9\)A given node or group can belong to several groups; when regrouping nodes, if several definitions are available, the user is prompted for selection.

source control-button2-click on a node opens emacs with the program at the corresponding line of the source file.

annotate button3-click on a node/group allows the user to enter a small text describing the node/group; this text will be displayed instead of the node name, allowing for annotation.

refresh click on the refresh button asks for an update of the displayed nodes and edges (compute the dependences to be displayed) and refreshes the display of the graph (process by dot and load into Tcl/Tk); as this may take a few seconds, we prefer to have it as a manual request instead of a standard process done after each of the previous interactions; several ungroups or regroups can thus be done before any refresh.

The typical way to explore a program is to successively focus on different groups, from highest levels that give a global overview of the program to the lowest ones that show the computations with more details: at every level, the dependences can be analyzed, nodes and groups can be annotated and new groups can be defined, allowing for the refinement of the initial hierarchy.

4. Evaluation

We tested our system with several programs in order to tune our interface and to compare the three different decompositions we introduced. We focus here on this latter question.

We compared qualitatively the benefits of the three decompositions with several programs of small size – a few hundred lines of code each – and we compared them quantitatively on these programs and two additional programs of medium size – about 3000 lines of code –; other tests are currently under work that anyway tend to confirm the results we obtained from the smaller programs. We present these preliminary results below.

The function decomposition is the closest to the structure of the program as perceived by the programmer. Actually, it clearly shows the real decomposition of the program into functions, as anybody can see it in the source files. It is thus very easy to recognize parts of the program at a glance. Furthermore, it allows to easily identify which variables a given function uses and/or modifies, as well as where these variables have been previously set and/or where they are used afterwards. For example, in Fig. 12 we see that three functions are called, namely init_tabbash, read_file and statwords. It is also easy to understand the global role of each one, even without looking at the source files: init_tabbash initializes the variable tabbash, read_file uses fp, nbword, tabbash and occur to set words and occur – this last one being thus modified during the call – while statwords uses
words and occur and doesn’t set any variable that would be used elsewhere.

The drawback of this decomposition is that it fails at showing clearly that some variables are set iteratively and thus that their values are used either during the loop or after the loop. For example, variable i is iteratively set in node #12 and iteratively used in node #10, #11 and #12 during the loop – its current, thus temporary, value being used – while variable occur is iteratively set in node #11 and used inside function read_file after the loop – its final value being thus used.

With the loop decomposition, on the contrary, one can either consider the outline of the loop or look at the detail of the content of the loop. For example, looking at Fig. 13 one knows that the loop uses an initial value for variable i and that the final value of occur is used after the loop; if one wants to examine the detail of the loop, one can focus on the content of the loop where one knows that every use and/or modification of variables is done during the loop, excepted those done in the encompassing main group (see Fig. 14).

On the other hand, with the loop decomposition, it is less easy to identify subparts of a program at first glance, as programmers are often less familiar with the data-flow structure than with the function names.

That’s why the function-loop decomposition seems to be

---

10 Usually, this means that the function displays or stores values.
the best one: even if it relies on two different structures that might become a source of confusion, it actually offers the benefits of the two others, easy recognition through groups with function names and iteration identification through groups representing loops. Fig. 15 shows the function-loop decomposition of the same program as discussed in Fig. 13: here the loop can clearly be identified, thus it is clear that occur used in read_file and statword is the final value, after the loop. The details of this loop have been shown in Fig. 14.

The superiority of the function-loop decomposition is also supported by the quantitative analysis we performed on a small program of 400 LOC, c-occur, and two programs of medium size, mandel [14] (we analyzed a sub-part of about 2500 LOC) and cs [2] (about 4000 LOC after macro-expansion). Actually, if we consider the number of nodes grouped in each group, the function-loop decomposition has the best average, as can be seen in Fig. 16. Furthermore, the proportion of big groups (number of nodes superior to 30) with respect to smaller groups is reduced with this decomposition while the size of each of these groups is reduced too. For example, in cs, only 4% of groups have more than 30 nodes with the function-loop decomposition, whereas the function, resp. loop, decomposition have 16%, resp. 7%, of such groups.

Therefore, with the function-loop decomposition we obtain in most cases a number of nodes per view that is small enough to allow the dependence graph to be rather easily analyzed. Nevertheless, we also observed cases – as well in groups with a high number of elements as with a small one – where complementary groupings were well founded. For example we decided to group together:

- strings of nodes where each uses one variable to update one other variable; the corresponding computation is well represented by a group using one variable – the first used in the string – to update one other variable – the variable updated by the last node of the string.

- triples formed by the test of an if, a node in the then branch updating a given variable and a node in the else branch updating the same variable; such a triple can globally be seen as an update of the given variable.

- sets of nodes that altogether produce a value for a single variable that is then used in multiple places – or interleaving computations [12].

- two (or more) sets of nodes that produce different values from which one is then chosen to produce a final value; the sets of computations can thus be grouped with the test.

Our interface allowed us to create easily the corresponding groups, but by hand. Therefore we focus now on elaborating a method to automatically build these new kinds of decompositions, that are more computation based than structure based, as are the decompositions we are able to build automatically at the moment. For this, further investigation, with programs of different, especially bigger size, is needed to identify a list of patterns of computations that should be grouped together and to define a strategy to automatically build them.

5. Conclusion

In this paper, we presented an approach to decompose dependence graphs of programs in order to have graphs of manageable size. We implemented a system that builds three possible decompositions to be browsed and analyzed through a graphical interface.

We tested our system with several programs of small to medium size. This showed that one particular decomposition, namely function-loop decomposition, seems to be the best one since it provides the smallest average number of nodes by group and the smallest proportion of groups of big size.

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>c-occur</th>
<th>mandel</th>
<th>cs</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>105</td>
<td>966</td>
<td>2364</td>
</tr>
<tr>
<td>function</td>
<td>9</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>loop</td>
<td>10</td>
<td>42</td>
<td>17</td>
</tr>
<tr>
<td>function-loop</td>
<td>6</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 16. Average nb of nodes per group
We now plan to pursue our research in the following directions:

- Enhancement of our GUI: an easier way to navigate inside the levels of the hierarchy as well as links between the nodes and the corresponding code have to be implemented. The detail of each computation should be directly accessible from each program point.

- Enhancement of the group representation: currently, a given node belongs to one and only one group. In case of function calls, this means that multiple calls to a given function are represented with multiple groups and this is counter-intuitive. We are looking for ways to handle multiple instances of the same group.

- Testing with programs of bigger size. This will show if the decompositions we propose are able, as we think, to produce groups of manageable size. Our conviction is that the size of the groups doesn't relate to the size of the program, but rather on the (stylistic) structure of the program; a large number of tests is clearly required to identify which factors have the biggest influence.

- Additional decompositions need to be defined and implemented (see Section 4).

Applying our tool to different kinds of programs will clearly allow us not only to tune our methods and algorithms, but also to evaluate its effectiveness for the detection of extra computation.

Acknowledgments

Many thanks to Giulio Antoniol for having taken time to run his system on a great number of programs, providing us with the basic material for our study.

References