

# Identifying the Information Needed to Understand Programs: an Experimental Approach

Françoise Balmas

Harald Wertz

Dépt. Informatique

Université Paris 8

France

{fb,hw}@ai.univ-paris8.fr

## Abstract

*In order to get a better understanding of what program understanding means, we are conducting experiments to observe programmers reading programs. Our primary concern is to identify which kind of information programmers are likely to search for in the code, to skip over or to miss. Secondly, we are interested to know how people construct their understanding of the program from the information they get and how they represent it, as we are convinced that this would help us to build automatic program understanding tools.*

*Based on first observations, we identified a preliminary list of useful information to provide programmers with and we constructed prototypical programs that compute and display this information. This process applies iteratively, as each new information view helps us to observe further how program understanding is built, therefore to construct further tools.*

*In this paper, we describe the experiments we did along with the programs we implemented on the basis of our observations, we outline the ways in which information and views are built, and we evaluate the information they provide as well as our approach to understand program understanding.*

## 1. Introduction

Understanding understanding is a difficult endeavor. It implies that we know what kind of knowledge is prerequisite to a given understanding task, what kind of processes and representations are activated during understanding, what kind of tools are necessary or what kind of expectations are driving the understanding process.

Following a similar methodology as [18, 22], in order to get a better understanding of what programmers do (and

what an automated program understanding system should be able to do) when they try (it tries) to understand a program, we are currently conducting a set of experiments combining the observation of programmers during different understanding tasks with the construction of prototypical tools that compute and display information the programmers wish to have at their disposal. We designed a set of four different understanding tasks:

- reading a program in order to find out what it does and how it works,
- reading a program in order to identify a small error and to correct it,
- reading a program in order to find out how to add another functionality,
- reading a program in order to modify its behavior.

Clearly, the first task distinguishes itself from the other three in that it does not imply a specific goal other than being able to describe the programs behavior. It is a kind of task which, in part, is included in all the others. Our hypothesis is that approaches to program understanding vary widely depending on the needed understanding: while the first task requires building an understanding of the entire program, the second task "only" requires an understanding of the part containing the error and the interaction of this part with the rest of the program, the third and fourth task require mainly an understanding of the underlying data-structures and of those parts of the program which interact with these data-structures. Our observation is that the approaches to program understanding also vary extremely widely depending on the previous experiences which seem to give the basic analogies on which the programmers build up their expectations which – finally – guide them in their understanding process (and which also, sometimes, create additional hurdles since they evoke "wrong" expectations).

Nb of lines	2704
Nb of files	7
Nb of include files	8
Nb of functions	37
Nb of modules	6

Figure 1. Characteristics of *cs*

In this paper we concentrate on some tools derived from the observation of programmers during the first task: reading of a medium sized program in order to understand what it does and how it functions. We will address the remaining tasks in another paper. Section 2 describes the experiments and the test program. In section 3 we give an account of our observations and the data and data-structures the programmers needed in order to understand the program. Section 4 describes the derived tools while section 5 gives a preliminary evaluation of the use of those tools. We conclude with further research perspectives.

## 2. Experiments

In this section, after presenting the program we asked our subjects to understand, we describe the two experiments our subjects performed.

### 2.1. The program

For the experiments reported in this paper we used a C program called *cs* that is designed to interactively examine C source code [6]. In fact, it first constructs a knowledge base of the program under scrutiny that then can be explored through an interface using the *curses* package.

The main quantitative characteristics of *cs* are summarized in Fig. 1 and clearly show that it is a medium sized program. Its architecture is rather easy to discover, as each module is in a separate \*.c source file along with its corresponding \*.h include file. Only one module relies on two such pairs and one include file is common to every module.

We believe it to be a good sample for our experiments firstly because it is short enough to be globally understood in a couple of hours – about the time we gave our subjects for the experiment – and secondly because it is not as simple as it might seem at first glance:

- many functions are several pages long,
- indentation is unusual,
- if many comments are well placed and very useful, others tend to mislead the reader,
- the use of hacks confuses the reader while trying to identify the goal of a unit of code,

```
void update ( parameters ) {
    /* declarations */
    dirscan( parameters );
    for (i=0; i

```

Figure 2. Sample of global variable use

- the use of global variables in different modules acting as transmitters of information is often hard to understand.

In Section 3 we will further discuss these features.

### 2.2. The experiments

In our first experiment, we asked 5 programmers, (ourselves included<sup>1</sup>) to understand the functionality of the program, to the extent they wanted and in the way they wanted. One was a novice C programmer, while the others were experts. Each session lasted a couple of hours.

At this time, we were merely interested in observing:

1. which kind of information programmers would search for,
2. how they would relate it,
3. how they would represent their understanding of the program,

rather than in analyzing program understanding strategies. That's why we gave our subject such an open task to perform. As Section 3 will show, this experiment gave us feedback on the first question, but not yet on the two others.

The second experiment began when one of us noticed that some global variables were used in such a way that it was impossible to get a global understanding of the role of a function without looking in detail at its sub-call graph, as it used a global variable that was initialized somewhere in these called functions.

Consider the example given in Fig. 2. The function *update* first calls *dirscan* and then tests both *srcs* and *scr*. As these variables are neither parameters nor local, but global variables, one must look at the definition of *dirscan* to identify what kind of values they may store. Then again, looking at *dirscan* is not enough and one has to look in detail at a third function, *updateentry*, to finally understand the purpose of *src* and *srcs*. Note that these two functions, *dirscan* and *updateentry*, are not trivial, are rather lengthy and require a lot of effort to be fully analyzed.

<sup>1</sup>Actually, the program was unknown to us when we began the experiment.

This kind of problem happens many times in `cs`, as several global variables are initialized or modified deep down in the tree of function calls and then used elsewhere, in the same module or even in another, may be also deep down in another tree of function calls. That's why we entered an iterative experiment process, where:

1. we implemented scripts to help analyze the use of these global variables and the way they convey information from one point to another,
2. we used the results provided by these scripts to further analyze variables and discovered new needs that led to new scripts,

and the process still continues today.

Actually, this experiment helped us to design the implementation of other useful scripts to analyze the role of global variables, but as the next section will show, further work has to be done on this topic.

### 3. Observations

In this section, we present the observations we have collected during our experiments and describe the information needs that we identified along with the scripts we implemented to fulfil them.

#### 3.1. First experiment

During the first experiment, where programmers were asked to understand the `cs` program speaking aloud, we observed the following behavior with respect to information searched for or missed:

1. One programmer searched for the routines of the programs, namely those functions that call only library function and no other user-defined function.
2. On several occasions, programmers tried to identify if a given function was a library function or not; actually, they were not all familiar with the `curses` library, thus found hard to distinguish, for example, between `initscr()`, a `curses` function, and `printlines()`, a user-defined functions. To get the answer, they had to use `grep` or the search facilities of `emacs` to find possible function definitions<sup>2</sup>.
3. Several programmers, confronted with a new function they had decided to analyze, first had a global look to the called (user-defined) functions in order to get an insight of the role of the given function, relating to the fact that, in `cs`, the functions are rather judiciously named.

---

<sup>2</sup>Indeed, `grep` has been recognized as one of the most used tool during maintenance activities [19].

4. In order to get a better understanding of parameter values, programmers searched several times for the functions calling a given function. Again, `grep` or similar search facilities were the chosen solution.
5. In many contexts, programmers were confronted with variables that were already initialized and they wanted to know how this had been done. For parameters, they had to look at the calling functions, while for global variables they had to search for other usages of this variable, identify the control-flow of the program and possibly find out where, and thus how, the variable had been initialized.
6. Some programmers encountered difficulties in identifying which computations were performed in which context in a function using several embedded complex control structures. For instance, in the `main` function, a `switch` is embedded in the default case of a surrounding one, while the previous case doesn't end with `break`. Therefore, the two last cases of the first `switch` are handled in the second, a construct clearly confusing readers.
7. Two programmers took a very long time to identify where a given treatment they expected to find was performed. Actually, from the manual page they had read, and from their own programming knowledge, they expected a source code examiner program to have a parsing phase. When looking at first glance at the beginning of the program, they didn't recognize any such treatment. After a deeper analysis and a rather precise identification of where this treatment should occur (for instance in the `update` function), again they still needed a long time before finally recognizing the call to `parsefile`. This is due to a very unusual indentation of the code, the call being performed when a given condition is verified and the whole expression being written on a single line:

```
if (condition) parsefile(arguments to the call);
```

that in addition was the last lines of the function. The two programmers had skipped over the lines, without reading them attentively, believing *that it was just a not very important test*.

Based on our observations, in order to help programmers handle understanding activities, we propose to provide them (at least) with the following information<sup>3</sup>:

**list of user-defined functions and global variables** This clearly helps distinguish the kind of functions or variables. Our visualization interface provides these two

---

<sup>3</sup>See Section 4 for a description of the way we compute it.

lists, allowing further exploration of it (see Fig. 13 and below).

**call graph** Obviously a call graph is necessary for understanding activities. We computed it in order to avoid library function calls, and to include information about the context in which a call occurs. This latter is done using a notation that describes the context in the following way:

- in every function statements are numbered,
- alternatives are noted with T, resp. F for the *true*, resp. *false* branches,
- in the same way, loops are noted with L.

Thus, for example, the second statement of the *false* branch of an alternative that is the first statement of a loop, itself the third statement of a function is noted 3.L.1.F.2.

A sample call graph is given in Fig. 3. It shows that function `findfile` calls `printline`, that in turn calls `xrealloc` and `xmalloc` two times, and function `getline` that also calls several times `xmalloc` and `xrealloc`. The context, even if obfuscating at first glance, allows one to know that the calls in `findfile` are performed only when a condition is true, 5.T.\*, inside a loop, 5.T.2.L.\*, when another condition is also true, 5.T.2.L.1.T.1. The called functions, `printline` and `getline` present a rather complex structure, as each of their calls are within alternatives branches and/or loops.

In order to more easily analyze the context of the function calls, we propose a restricted version of the call graph where only the called functions (callees) appear, and not the sub-call graphs (see Fig. 4). Here, we clearly see that the two calls happen in the same context, which means that the second is an argument of the first one.

Alternatively, we propose a call graph including only calls *to* a given function, allowing for the examination of its caller's contexts.

findfile	->	printline	5.T.2.L.1.T.1
printline	->	xrealloc	7.F.1.T.4
printline	->	xmalloc	7.F.3
printline	->	xmalloc	7.F.5.T.1
findfile	->	getline	5.T.2.L.1.T.1
getline	->	xmalloc	8.T.1.T.3
getline	->	xmalloc	8.T.1.T.5
getline	->	xrealloc	8.T.1.T.10.L.3.L.1.T.1.T.4
getline	->	xmalloc	16.T.3
getline	->	xmalloc	26

Figure 3. Sample call graph

findfile	->	printline	5.T.2.L.1.T.1
findfile	->	getline	5.T.2.L.1.T.1

Figure 4. Call graph restricted to callees

13.T.31.L.36.T.11			
main	- lines / l ->		findfile
main	- field / literal ->		findfile
13.F.1.T.1.L.3.F.11			
main	- lines- / l ->		findfile
main	- query[1] / literal ->		findfile

Figure 5. Sample data-flow graphs

**data-flow graph** For further analysis of calls, a data-flow graph is needed. We provide such a graph for every call of the program, giving the value transmitted in arguments as well as the name under which it is used in the called function. For example, Fig. 5 shows the data-flow graphs of the two calls to `findfile`. This allows one to recognize that the first argument is the same in both cases while the second differs.

Note that if sub-functions were called, further data-flow information would also be given.

**simple variable usage** To help identify where global variables are initialized and used, we provide, for a given variable, the list of the functions that either Use, Produce or Use/Produce the variable because they either read, write or read/write the value of the variable. Fig. 6 gives an example.

If this is useful information to help perform understanding tasks, it is insufficient in many contexts. That is the reason for our second experiment which focuses mainly on variable usage analysis.

Uses	cpp	ofp
Uses	indexident	ofp
Uses/Produces	parsefile	ofp
Uses	string	ofp
Uses/Produces	update	ofp
Uses	writerec	ofp

Figure 6. Simple variable usage

ident	[readrec / st]
M	Store str 1.L.24.F.1
line	[readrec / line]
M	Store 0 1.L.14

Figure 7. Modifications of variables

### 3.2. Second Experiment

During this experiment, we were first interested in understanding the effect of a function call, in order to identify which variables were initialized or modified during a call. As we built scripts to provide the necessary information, we identified new needs that led to new scripts, and so on. We describe this iterative process below.

**Modification of variables** Our first view of a call accounts for all the modifications that are performed during the call. Fig. 7 shows an excerpt of what we get for the call `main->findlit`. It first gives the name of the variable and the function in which the modification occurs, with the name under which this variable is handled in the function in case of a parameter. The following line describes the modification itself: we use letters to characterize the kind of modification (for example, M means the Modification of a pointer/array, N a numerical accumulation and U an Update of a non-pointer variable) and the rest of the line gives the value of the modification. In the first example of Fig. 8, the value of variable `str` is stored in pointer `ident` within the function `readrec` where it is named `st`. The third line gives the actual context in which the modification occurs.

This view provides useful information about the modifications of variables performed during the call – actually all are shown – but it also provides too much information: when we want to know the effect of a call, we would like to be aware of only those modifications that have an effect on the rest of the program.

**Use and modification of variables** The second view we constructed collects, for a given call, not only the modifications (in a compact manner) but also the uses of any variable. The letters G and A are intended to distinguish between usages of the Global variables and those of variables received as Arguments. The example of Fig. 8 shows these different usages for the call `main -> findlit`.

With this information, one can know if a given variable is used at some point of the program. But to precisely

UsesG	srcs src linemode line ident
UsesA	lines line indexfile ident field
ProducesA	lines line ident

Figure 8. Uses and modifications

Uses/Produces	findcallee	ident
Uses/Produces	findcaller	ident
Uses/Produces	findid	ident
Uses/Produces	findinclude	ident
Uses/Produces	findlit	ident
Uses/Produces	findposix	ident
Uses/Produces	update	ident

Figure 9. Restricted usage

know if, say, variable `ident`, modified during the call to `findlit`, is used elsewhere, one has to browse all the call views.

**Restricted usage** To avoid such a tedious process, we built a kind of abstraction of the former views. For a given function, `main` in the example cited, we followed the use/modification information chain from the function where it occurs to the functions called by `main`. In Fig. 9, `ident` is noted as used in `findcallee` while it is actually also used by functions called by `findcallee`. In the same manner, `ident` is only used in `findposix` but modified in functions called by it.

We found this view very useful in many contexts where it allowed us to stop considering variables that were no longer used or modified after the call we were analyzing. But, unfortunately, we found cases where it was not efficient because it was too simplified: in this view, we can only know that a variable is used or modified, we don't have any insight into the order in which the usages occur. Actually, if we could identify that a variable is always modified before it is used again, we could also stop analyzing it.

From the latter, it seemed clear to us that some global variables were locally used – they were global just for technical reasons – while others were truly global and conveyed data from one point of the program to another. In order to distinguish between these two cases of global variables, we developed the following two views.

**Function panorama** This view lists all the events of a function, namely the variable use or modification, the function calls and the returns of values. They are given in an order corresponding to the order of evaluation of

findposix	(l indexfile)	
TestsA	l	1
...		
PRIM	fopen	5
	- no argument	
UsesA	indexfile	5
U-L	fp (fopen indexfile tmp_string)	5
CALL	myendwin	5.T.1
	- no argument	
PRIM	fprintf	5.T.2
...		
UsesG	ident	7
CALL	readrec	7
	- fp -expr- tag device inode line fun ident	
	UsesA : line inode ident fp device	
	ProducesA : tag line inode ident fun device	
...		

**Figure 10. Panorama of a function**

the program, along with the contexts in which they occur, both paths of alternatives being given in the same order as in the code. This allows us to follow precisely how variables are used, as we get the events in the right order, inside a single function.

For example, in Fig. 10, we can easily analyze the usage<sup>4</sup> of `ident` in the function `findposix` (it is used in statement 7 as argument given to function `readrec`) but we don't have any way of knowing how it is used during the call to this function, except in examining the corresponding panorama.

**Slice** This last information is an attempt to combine the previous one with information on calls. It presents the list of all the variable usages in the order they actually occur within the program: each modification is detailed as in the *modification of variables* view, variable uses are listed too and in addition the actual point in which the usage occurs is given as two paths, the function path `f1/f2/.../fN_where_usage_occurs` and the context path that gives the corresponding context `call_context_in_f1/call_context_in_f2/.../usage_context_in_fN`. As this information takes up a very huge file, it is viewed through three different slices:

- all the usages that concern a given variable,
- all the usages that concern a given function call,

<sup>4</sup>Here `Tests` indicates a special kind of `Uses` where the variable is tested; `G`, `A` and `L` indicate usages of resp. Global variables, Arguments and Local variables; `U` and `M` are intended to identify the kind of modification of the variable, an Update and a Memorization in an array/pointer; `PRIM` and `CALL` denote calls to resp. primitives and user defined functions, these last being annotated with their corresponding uses and modifications of variables.

- all the usages that concern a given variable during a given function call.

Fig. 11 gives an excerpt of the slice for variable `ident` during every call to `findposix` in function `main`. It presents all the usages of `ident`, during `findposix` and the functions called by it: we see again that `ident` is used in `findposix` in statement 7, where it is given as argument to `readrec`, and that is then used in function `readrec` within a loop when a given condition is true while modified when the same condition is false. This allows one to understand that if a call to `findlit` modifies the value of `ident`, this value has an effect on all subsequent calls to `findposix`.

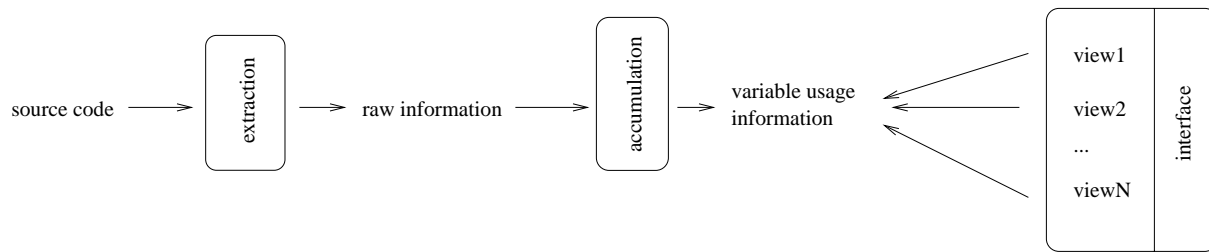
Path	/main/findposix
Loc	/13.T.31.L.36.T.29/7
UsesG	ident
Path	/main/findposix/readrec
Loc	/13.T.31.L.36.T.29/7/1.L.24.T.1
UsesA	ident [ st ]
Path	/main/findposix/readrec
Loc	/13.T.31.L.36.T.29/7/1.L.24.F.1
ProducesA	ident [ st ]
	M-a Store str
Path	/main/findposix/identcheck
Loc	/13.T.31.L.36.T.29/7.L.2.T.11/2
UsesA	ident [ identifier ]
Path	/main/findposix/identcheck
Loc	/13.T.31.L.36.T.29/7.L.2.T.11/2.T.1
UsesA	ident [ identifier ]
Path	/main/findposix/identcheck
Loc	/13.T.31.L.36.T.29/7.L.2.T.11/2.F.1
UsesA	ident [ identifier ]
...	

**Figure 11. Call slice**

These different views allowed us to analyze the role of the global variables, then to identify how data is conveyed as global variable inside the program, but currently we can only do it by hand simulation, examining the views, and could not yet automate this discovery process as much as we intended (see Section 5).

## 4. Information construction and visualization

In this section, we outline how the data previously mentioned is computed and introduce the interface that enables us to visualize it.



**Figure 12. Architecture of the tool**

#### 4.1. Data construction

Existing tools like control- and data-flow builders, or program slicers, would allow us to get most of the information we mentioned in the previous section. We chose however to implement our own for the following reasons:

- such tools are not all accessible, or we cannot run them on our platform,
- most are separate tools, the CANTO environment being a noticeable exception [1],
- we wanted to be able to modify or extend them and this could conceivably require a too much investment with programs written by others.

We therefore implemented prototype scripts to extend our existing tool (see below) so that, even if it is not very efficient, it enabled us to perform our experiments (see Section 5).

The global architecture of the process is given in Fig. 12. The source files are first processed to extract raw information about each function of the program. For this first step, we reuse the system F. Balmas had implemented in order to compute outlines of loops [2]. This system transforms C code in a Lisp-like notation that can then be handled by a special-purpose Lisp evaluator. This evaluator processes the code as a normal evaluator would, but instead of computing results for function calls, it produces information about the performed computations:

- initialization and modification of variables,
- use of variables,
- function calls,
- arguments to function calls,
- returned values.

Each information is given along with the notation that describes the context in which the computation occurs (see Section 3).

In the second step, data on variable usage is accumulated using control- and data-flow information in order to describe the effect of every function call. This is done with AWK and BASH scripts that read the raw information previously extracted while simulating function call evaluation along with their arguments. Thus this process accumulates variable usage information, with actual variable names, from every function call view point.

Variable usage is stored in three different formats:

1. The list of the variables that are read, distinguishing between the global variables and those that are given as arguments (*UsesG* and *UsesA*) and the same for the modified variables (*ProducesG* and *ProducesA*).
2. The list of the variables that are modified along with details about the actual modification: the function in which the modification occurs, a keyword to describe the kind of modification (see section 3.2) and the value given to the variable.
3. The list of the variable usages, in the order they actually occur within the function call with each usage detailed as in the previous format. In addition the actual point in which the usage occurs is given as two paths, one for the functions, another for the contexts (see in section 3.2 the paragraph on slices). In the global execution of a program, such a format represents a slice for a particular call.

#### 4.2. Visualization interface

Our tool presents itself as an extended cross-reference on functions, variables and function calls, that allows one to visualize<sup>5</sup>, in different ways, the data described in the previous section, according to the views we described in Section 3. It is a Tcl/Tk interface with radiobuttons to choose the desired view and listboxes to select the desired function, function call or variable. The main window is shown in Fig. 13.

For a function, the possible views are:

<sup>5</sup>The views are built by extraction and/or abstraction from the variable usage information described above.

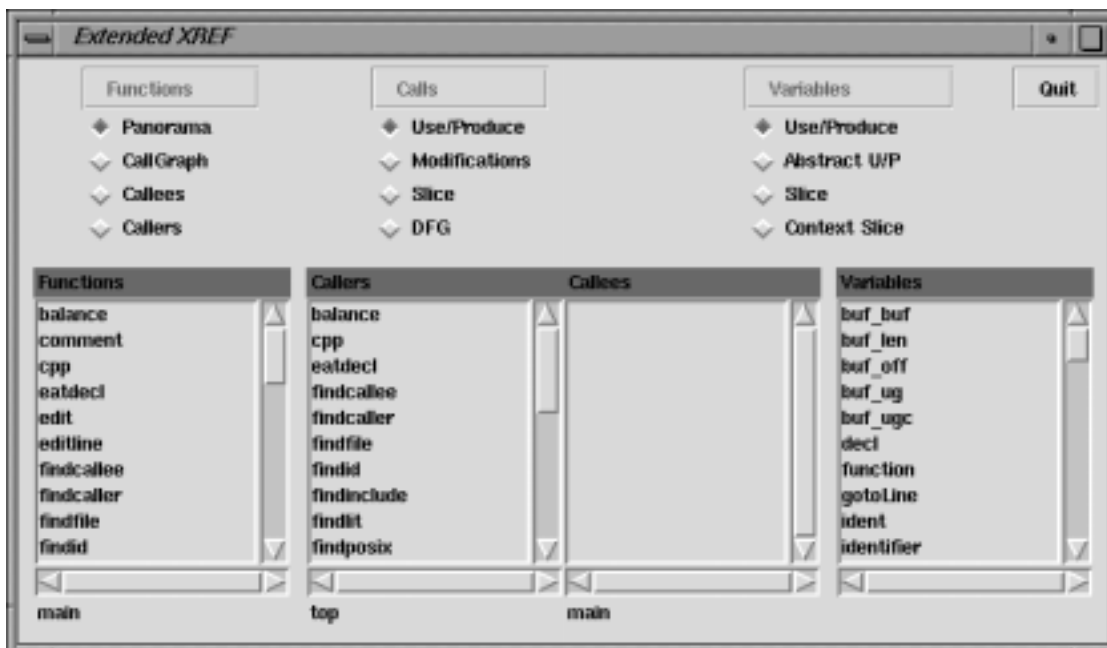


Figure 13. Main window

- the panorama of the function, that lists every event of the function with its context;
- the call graph, in a textual form, with the context of each call;
- the list of the called function, with the context;
- the list of the calling functions, also with the context.

For a call, the main possible views describe the effect of a given call, in the three format described in the previous section. The data-flow graph, in a textual form, can be visualized as well.

Variables may be viewed in the following ways:

- the variable usage,
- for a given function, the list of its called functions that use or produce the variable (restricted variable usage),
- a slice of the variable usage in the whole program,
- a sub-part of such a slice, namely a restriction to a given function call.

Note that several of these views have an alternative format, in the RIGI Standard Format [27], that can be displayed and manipulated in the RIGI environment.

## 5. Evaluation

We designed our tool as a prototype to evaluate the information needs we gathered during our experiments. Therefore, we implemented simple algorithms which later will also use such advanced techniques as pointer aliasing, for data flow or for dynamic control flow [7], def-use dominance relation [11], symbolic evaluation [25] or partial evaluation [10]. Integration of those techniques in our tool will make it a practically useful program understanding tool.

Nevertheless, it allowed us to evaluate our experiments. Basically, it has been used by an expert programmer, involved both in academic research and industrial developments, whom we asked to:

- perform our first experiment, namely understanding the `cs` program, using the information provided by our tool,
- discuss the information we provide with respect to the information he would like to be provided with during his maintenance tasks.

Following are the main features he pointed out positively:

- Many types of information, both on functions as well as variables and function calls, are integrated in a single tool that can easily be extended; the drawback of the existing tools, as already mentioned, is that most

compute only one kind of information, thus a programmer has to switch back and forth between many tools to get the information wanted.

- The restricted variable usage view introduces a highly useful aggregation mechanism that allows one to get a more compact view of, or another view point from the usual variable usage view. This is a local – and partial – response to the question of *how much information has to be displayed?* (see below).
- Much of the information in our tool, such as information on function calls, variable usages or call effects, are given with an indication of the context where the event occurs. Even if the notation in which the context is given is somewhat obfuscating, it is nevertheless a good indication of the global structure, thus the complexity of the function<sup>6</sup>. In addition, it helps correlate the different events as it introduces information on the sequencing between them.

Below are his suggestions for further enhancements or extensions of our tool:

- To compute also for local variables the same set of information as the one computed for global variables, that is: considering each function as a global context.
- To bind the context indications, in every kind of view, to the corresponding line in the source file. This will be a straightforward extension of our tool.
- To add the possibility of getting slices of two or more variables together. This is intended to help detect if, and how, sets of variables are used together, to identify which values they receive and when, and to prevent dead lock situations: ideally, the tool would itself detect such critical situations!
- To investigate further how information has to be presented, either visually or textually. Actually, our views are pretty-printed with rather simple rules, that often lead to too much information. For example, the Panorama view contains too much information as soon as the function is more than about ten lines long and becomes quite more unreadable than the code itself<sup>7</sup>. A better disposition within the window or some way of highlighting given information would greatly improve the usability of the information.
- To investigate further the amount of information to display so that it is neither overloading the perceptive capacities of the user, nor hiding too much information.

---

<sup>6</sup>Actually, the context notation can be transformed into a structured graph.

<sup>7</sup>But note that D.E. Knuth argues that *no* function should be longer than 10-15 lines [9].

This is, as we know since Miller's seminal paper [12] a very difficult problem, with which all toolbuilders are confronted. Thus, we too were often confronted (see Section 3.2) with the problem that a first view gave too much information, indicating the need for a restricted or abstracted view, which, once constructed, didn't give enough details any more. Thus, automatically choosing the right level of granularity seems to be a very hard and open question; we would like to have a dynamically extensible tool, in which the programmer could define exactly the level of detail s/he wants to see as well as the abstraction s/he wants to get. A first step in this direction is outlined in [14].

Clearly, the last two points refer to still open questions requiring further research (see also [23], [16] and [8]).

## 6. Discussion

Program understanding has been object of interrogation and research for quite some time. Winograd, with his notion of *complexity barrier* [26], first identified the general complexity of any understanding task, while Brooks introduced still open basic questions about practical representations of programs [3].

Early program understanding research mostly focused on comprehension strategies [4, 13, 17, 20]. More recently, [24] proposed a code comprehension model integrating these strategies; [21] explored the construction of software visualization tools; [15] experimented *layered explanations* as a program comprehension methodology; [5] analyzed the knowledge units required to understand programs.

In this paper, we have introduced our approach to *understanding program understanding*, namely observation of programmers reading code as an attempt to identify the information they search for, how they organize it and finally the way they represent their understanding of the program.

In this context, we performed two experiments that allowed us to identify:

- a preliminary list of information necessary to understand programs,
- a list of views which help understand the effect of function calls as well as the role of variables.

In order to evaluate this information, we implemented several scripts that compute information about programs and we designed views that allow visualizing it, using extraction and/or abstraction. These views are accessible through an integrated tool that presents itself as an extended cross-reference of functions, global variables and function calls.

Though our tool is still a prototype that uses simplified algorithms to compute information, it has already been successfully evaluated by experienced programmers. It demonstrates the practicability of our approach: from observations made during experiments, we could implement a tool that provides useful information for program understanding and understanding program understanding; it will help us perform new experiments – analyzing programs using different programming techniques and styles, focusing on different questions about programs – that in turn will identify new information, and so on. In the long run, we expect that this approach will allow us to understand enough program understanding to be able to implement better automatic program understanding tools.

**Acknowledgment:** We would like to thank Touria El Mekki, Patrick Greussay, Jean Méhat and Damien Ploix for their noticeable contributions to our experiments. The English used in this paper has considerably benefited from the advice and the careful reading of Ellen Sparer.

## 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. Outlining C loops: Preliminary results and trends. In *Proceedings of the 5th Working Conference on Reverse Engineering*, Honolulu (Hawaii), 1998.
- [3] F. P. J. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading (MA), 1975.
- [4] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. J. on Man-Machines Studies*, 18, 1983.
- [5] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Proceedings of the 5th Working Conference on Reverse Engineering*, Honolulu (Hawaii), 1998.
- [6] <http://www.moria.de/michael/cs/>.
- [7] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. Points-to analysis for program understanding. *Journal of Systems and Software*, 44(3), Jan. 1999.
- [8] C. Knight and M. Munro. Visualising software – a key research area. Technical Report 5/99, University of Durham, Durham (UK), 1999.
- [9] D. E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford (CA), 1992.
- [10] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In E. C. Berkeley and D. G. Bobrow, editors, *The programming Language LISP: Its Operation and Applications*. Information International, Inc., 1964.
- [11] E. Merlo and G. Antoniol. Def-use dominance relation and its application to unconstrained def-use intra-procedural computation. Personal communication, 2000.
- [12] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, (63), 1956.
- [13] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 1987.
- [14] D. Ploix. *Elaboration, Réalisation et Evaluation d'un Environnement de Programmation Analogique*. Thèse de doctorat, Université Paris 8, Saint-Denis, 1999.
- [15] V. Rajlich, J. Doran, and R. T. D. Gudla. Layered explanations of software: A methodology for program comprehension. In *Proc. of Int. Workshop on Program Comprehension*, 1997.
- [16] S. Reiss. Visualization for software engineering – programming environments. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a multimedia experience*. MIT Press, Cambridge (MA), 1997.
- [17] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Inc., 1980.
- [18] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *Proc. of the Int. Workshop on Program Comprehension*, 1998.
- [19] J. Singer and T. C. Lethbridge. What's so great about 'grep'? implications for program comprehension tools. Research note.
- [20] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), Sept. 1984.
- [21] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proc. of the 5th Int. Workshop on Program Comprehension*, Dearborn (MI), 1997.
- [22] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the 4th Working Conference on Reverse Engineering*, Amsterdam (The Netherlands), 1997.
- [23] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Mueller. Programmable reverse engineering. *Int. J. of Software Engineering and Knowledge Engineering*, 4(4), Dec. 1994.
- [24] A. von Mayrhauser and M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proc. Sixth Int. Workshop on Computer-Aided Software Engineering*, Singapore, 1993.
- [25] H. Wertz. Stereotyped program debugging : an aid for novice programmers. *Int. J. on Man-Machine Studies*, 16, 1982.
- [26] T. Winograd. Breaking the complexity barrier again. *ACM SIGPLAN Notices*, 10(1), Jan. 1975.
- [27] K. Wong. *The Rigi User's Manual - Version 5.4.4.*, June 1998.