PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE

ESCUELA DE INGENIERIA

PROGRAM COMPREHENSION TECHNIQUES ANALYSIS ON NON-OBJECT ORIENTED SYSTEMS

DANIEL CÓRDOVA

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Advisor:

YADRAN ETEROVIC

Santiago de Chile, Julio, 2011

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

PROGRAM COMPREHENSION TECHNIQUES ANALYSIS ON NON-OBJECT ORIENTED SYSTEMS

DANIEL CÓRDOVA

Members of the Committee :

YADRAN ETEROVIC

SERGIO URIBE

STEREN CHABERT

ANDRÉS GUESALAGA

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Santiago de Chile, July, 2011

To my family, friends, and all my people.

ACKNOWLEDGES

I want to thank all the people that helped me in this work. First of all I want to thank Sergio Uribe for his active role in the development of my work as a domain expert. Also I want to thank Fernanda Campos and Esteban Cortázar for their contributions and previous work researching methods and techniques to help in the understanding of complex software.

I thank my teacher Yadran Eterovic for his support and help along my entire Master scholarship, and the teachers Pablo Irarrázabal and Cristián Tejos for their help in the birth of my Master project.

Finally I have to thank my family and friends, which have been the foundations of my achievements and goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

RESUMEN

Es sabido que comprenden código ajeno es una de las tareas que más tiempo consumen a la hora de actualizar o mantener software. Varios estudios se han hecho en ésta área llamada *Program Comprehension (PC)*, pero la mayoría han sido probados sobre pequeños sistemas Orientados a Objetos (OO). Este trabajo busca discutir el posible desempeño de algunas técnicas y herramientas de *Program Comprehension* en un sistema de tamaño industrial no OO (non-OO), y determinar qué cambios, de haberlos, son necesarios para usar técnicas de PC diseñadas para sistemas OO en otro tipo de sistemas.

Nuestros resultados evidencian que el enfoque automático-visual basado en investigaciones previas mantuvo su desempeño al ser cambiado de un énfasis OO a uno non-OO, pero el traspaso del enfoque automático-genérico actuó de peor manera en ambientes non-OO que en ambientes OO. Esto muestra que las herramientas probadas en OO existentes probablemente fallarán al ser aplicadas en otros tipos de paradigmas, pero la investigación detrás de ellas puede ser exitosamente aplicada si no es fuertemente dependiente de alguna característica específica del paradigma.

Palabras clave: Program Comprehension, non-object oriented, industrial-size software.

ABSTRACT

It is known that to understand foreign code is one of the most time-consuming tasks when you have to update or maintain software. Several studies have been done in this area called *Program Comprehension*, but most of them are tested on small object oriented systems. This work aims to discuss the possible performance of some *Program Comprehension* techniques in an industrial-size non-OO system, and then determine what changes, if any, are needed to use PC techniques designed for OO systems on other kind of systems.

Our findings evidence that the visual-generic approach based on previous research kept its performance when switched from an OO emphasis to a non-OO emphasis, but the port of an automated-generic tool acted in worst way in non-OO environments than in OO environments. This show that the existing tools tested on OO systems will probably fail when applied on non-OO systems, but the research behind them can be successfully applied if it's not strongly dependent on a specific paradigm characteristic.

Keywords: Program Comprehension, non-object oriented, industrial-size software.

1. INTRODUCTION

1.1 Motivation

Whenever a programmer is asked to understand a code, he creates a mental model of the system facing the source code with his general programming knowledge. The goal of *Program Comprehension* (PC) techniques and tools is to help the development of such mental models (Storey, Wong, & Müller, 2000), either identifying interesting parts of the source, showing in a more friendly way the dynamic or static information of the program, etc.

Following this line, there are many experiments and investigations that aim to make easier the understanding task. However, most of those works are focused on mid-size object oriented (OO) systems (Cornelissen, Zaidman, van Deursen, Moonen, & Koschke, 2009). This fact raises the question of whether these findings can be directly applied to real life software, i.e., an industrial-size system not necessarily written in an OO language.

In the context of this work, we identified two approaches used when developing or researching OO PC tools. In one hand we have works meant to a specific target language, as we see in the code browser tool Juliet (infotectonica sa, 2006) meant only for Java, and in the other we have works that aren't explicitly engaged to a specific language, but rather to OO environments broadly, as we see in the feature identification tool proposed in (Antoniol & Guéhéneuc, 2005).

The first kind of works is obviously not suitable to be ported directly to other paradigms, because it depends directly of the language it was meant for. But something interesting happens with the second kind of works. In the context of this thesis, we found that most of these works, even when they're not explicitly designed for an OO language, they still

use key assumptions that can only be met in this kind of environment. Looking again at Antoniol's work for example, we can see that the first stage of his feature identification relies on typical OO structures like classes, interfaces, etc; because he creates a model of the target program composed by these entities. Then, if we want to port this approach to other environments where such structures don't exist, we will necessary have to redesign at least the stage of program model creation, finding good replacements for these entities, and even if we do so nothing assure us that we will have a similar performance with our implementation.

This tendency repeats among most of the works that we reviewed. For example, in the automated concept identification method proposed in (Carey & Gannod, 2007), they use several metrics to create a vector to represent each class in the target system, that will be used later to define proximity between classes, leading to define clusters of classes that will represent a concept. Besides the fact that they use classes to divide the software, because it's not that hard to find a substitution of a class in other paradigms, the metrics themselves are based on OO concepts like the number of overridden methods in a class, the number of direct subclasses, etc. The problem when trying to port this work rises when you have to define new metrics to create the vectors of each class (artifact of code), because the original work succeeded with its metrics, and nothing assures that other metrics will have the same results.

However, there are some works that aren't necessary bounded to any specific OO assumption at all. That is the case of Cerberus (Aho, Antoniol, & Guéhéneuc, 2008) a tool for concern location that combines static analysis in the form of information retrieval (IR), dynamic analysis in the form of execution tracing, and an automated prune algorithm that feeds on the previous results to automatically extract features of a program. The IR phase consists on automatically extracting terms of all the static data (code, documentation, etc.) to create a starting set of possible concerns in the form of "terms", like the name of a method or class, or entities present in the documentation like

features or requirements. Then, in the execution tracing phase, the activated elements of the code are rescued to create a set similar to the one created on the previous phase. Finally, the combination of the two prior phases is used as a seed to the pruning phase, where more elements are added and some other are removed depending with their relation to the elements already found. As we said in the beginning of this paragraph, this works doesn't make any strong assumption related with OO concepts, but the study case used to validate their hypothesis is a system implemented in Java, which raises again the question if these kinds of novel approaches and findings can be successful on any other kind of environment.

Also, there are some works based on non-OO languages, but they are the minority amongst the research field, as the survey done in (Cornelissen, Zaidman, van Deursen, et al., 2009) stated, and almost all the commercial tools meant for PC that were reviewed in the contest of this thesis were either code browsers like (GrammaTech, 2007) or project management suites like (Intland Software, 1999).

The question if OO PC tools can be adapted to other paradigms is very important because, as we introduced previously in this same chapter, a big share of the PC research has been validated on OO systems, and most of the software used in the industry is non-OO; instead it is legacy, web based or any other kind. So, if we really aim to introduce PC in the actual industry, we should take care of the validation of PC tools developed with an OO study case in mind in other kinds of software, so all the efforts made until now have not just an academic value.

1.2 Object-Oriented versus non Object-Oriented

While it is true that the notation of the language (OO, procedural, declarative, etc.) does not necessarily defines the structure or quality of the final program, it does have an

incidence in the information we can retrieve from it. According to (Wiedenbeck S., 1999) and previous work presented in (Green T.G.R., 1984), programming paradigms highlights certain information at the expense of other information. Even if you have a clear goal and design, the platform you choose to implement your software will give you the guidelines of your final solution. For example, it's hard to create an efficient system with FORTRAN to solve a problem that requires a lot of communication between entities (hierarchy may not be enough to make a good design), but it's easy to do the same task with any OO language because the message passing between objects can handle all the communication in a simple way.

The authors of these works state that the OO notation highlights domain information and functionality due to its class notation, at the expenses of occluding data transformations due to its capacity to encapsulate the implementation details. On the other hand, procedural notation highlights operations and control flow due to its sequential representation (all the information necessary to perform an operation is together), in exchange of data flow and domain information due to the low level of abstraction that this notation allows.

These findings lead us to think that, independently of the programmer's skills and the design's quality, it is highly probable that a system developed in a determined language will have structures that follow its own highlights; in fact, these structures may have been the cause of choosing that environment in the first place. For example, procedural languages are generally preferred when the programmer needs to implement a system highly efficient in calculations, which is exactly the kind of information this notation emphasizes. Meanwhile, OO languages are preferred when the system that will be implemented needs a complex message-passing support, which can be achieved with the encapsulation and other mechanisms the paradigm offers.

From all the above we can derive that PC tools meant to specific languages or paradigms will perform better if they rely on these “information highlights” to help in the understanding, because the extraction of meaningful information about the system will be easier. But this can also mean that when a tool that takes advantage of certain highlights is applied in a different paradigm, that probably will not highlight the same information, there is a high chance that the tool may drop its performance because the information it needed to make its deductions isn’t as easy to collect.

As an example we will compare two programs that do exactly the same: implement a state machine driven by user input, one written in C and the other written in Java. These programs are based on the first example proposed in chapter 20 of (Meyer, 2004), where the author first states the problem to be solved, then proceeds to implement a solution in a structured pseudo-code, to finally implement another solution that improves the first one in a OO pseudo-code.

```
1  int execute_state(int s, int* n)
2  {
3        int answer; //user input
4        int ok; //boolean
5
6        do
7        {
8              display(s); //shows information according with the state s
9              read(s, &answer); //reads the user input
10             ok = correct(s, answer); //validates that the answer is correct
11
12             if(!ok)
13             {
14                   message(s, answer); //shows an error message
15             }
16       }while(!ok);
17
18       process(s, answer); //process the user input acording to the state s
19
20       *n = next_choice(answer); //next transition based in the user input
21 }
22
23 void main()
24 {
25
```

```
26        int state, next:
27
28        state = 0; //init state
29
30        do
31        {
32              execute_state(state, &next);
33
34              state = transition(state, next); //returns the next state
35        }while(state != -1); //final state
    }
```

Table 1: C implementation of the state machine example

This table shows the code in C for the structured pseudo language of the original example.

```
1   class State
2   {
3   public:
4         int input, choice;
5
6         void execute()
7         {
8               boolean ok;
9
10              do
11              {
12                    display();
13                    read();
14                    ok = correct();
15
16                    if(!ok)
17                    {
18                          message();
19                    }
20              }while(!ok);
21
22              process();
23        }
24
25        abstract void display(); //shows information according with the state
26        abstract void read(); //reads the user input
27        abstract boolean correct(); //validates that the answer is correct
28        abstract void message(); //shows an error message
29        abstract void process(); //process the user input acording to the state
30  }
31
32  class ExampleState extends State
```

```
{
      void read()
      {
            //read the user input
      }

      //similar for all the abstract methods
}

class Application
{
public:
      State associated_states[];
      int transitions[][]; //transition table

      int initial_state_index = 0; //initial state

      Application(int num_states, int num_transitions)
      {
            transitions                    =                    new
int[num_states][num_transitions];
            associated_states = new State[num_states];
      }

      void main()
      {
            State st;
            int st_number = initial_state_index;

            do
            {
                  st = associated_state[st_number];
                  st.execute();

                  st_number = transition[st_number][st.choice];
            }while(st_number != -1); //final state
      }
}
```

Table 2: Java implementation of the state machine example

This table shows the code in Java for the OO pseudo language of the original example.

Even when these two solutions do the same job and both are well enough designed, we can clearly see differences in the implementation due to the different tools that both languages offer. For example, we can see that the structured solution presents “high coupling”, this is, it contains in one function the logic to execute *display* in each state of

the system, plus the control flow statements needed to decide which is the actual state (the same for *read*, *correct*, *message* and *process*, that's why all those functions needs 's' as an input), but the OO example managed to distribute the responsibility amongst different classes, leaving the resolution of the actual state to the polymorphism; for example the method *read* is called in line 13, but its actual implementation can be in line 34 in a different class, not directly in line 26 in the same class State.

One particularity that arises due to the structured solution's design is that the variable 's' is transmitted through all the system, from the main loop of the application (line 31) to the different functions that are called in each state (lines 8 to 18) and finally inside those functions to decide which behavior must be executed, breaking the purpose of the function decomposition. This is not the case in the OO example, where the state of the system is needed only in the main loop (lines 63 and 64) because it's the language itself that is in charge of judging which behavior will be used thanks to polymorphism.

If we, for example, could test on these two examples a PC tool that claims that the most frequent called entities of a system (let it be classes or functions) are worth reviewing when doing a maintenance task, we could get absolutely different answers from analyzing the structured solution and the OO solution. If we pick an execution trace from the structured system, the PC tool will say that the most important functions in the structured case are *display*, *read*, *correct*, *message* and *process* regardless the sequence of states executed, because the calls in lines 8 through 18 will always be made, despite the state the system is.

On the other hand, if we could take an execution trace from the OO solution, the tool will say that the class State and only the classes that implements the methods *display*, *read*, *correct*, *message* and *process* that were effectively called after the polymorphism was resolved (for example the class ExampleState) are indeed the most important classes, leaving behind the classes representing the states that were less passed through.

Nevertheless, PC tools don't depend only in the target language; there is also the research that leads to the development of the tool and the algorithm that supports it. Since these elements are not directly influenced by the language itself, it may be accurate to affirm that they can be ported to any paradigm, though the implementation of a tool in this new environment shall be started from scratch.

1.3 Hypothesis

Our hypothesis is that PC tools developed for OO systems can't be directly applied to non-OO systems due to the paradigm change. However, the research and assumptions that supports those tools can be used to create new tools that are less dependent on the paradigm of the target language. We have to emphasize that we are not saying that is impossible to use PC tools validated in OO systems under other kinds of environments, but rather that the implementation can't be direct and that the change in the study case's structure may cause that the results originally met in the OO system can't be repeated in this new environment.

In order to prove this two-part hypothesis, first we will adapt an existing tool that aims to help in the understanding of an OO system to be used in our non-OO study case, and then we will develop a new tool based on previous research related to the understanding of OO systems.

1.4 Objectives

The main objective of this thesis is to design and make a field study of PC tools in a non-OO environment, comparing their contribution to the understanding and their usability with the performance the tools had in its original OO environment.

The specific objectives are:

- Adapt a PC tool meant for OO systems, to be tested on the study case. The adaptation will consist in recreate the tool, preserving its main algorithm and assumptions, but changing any direct dependency with OO it could have (like the analysis of specific keywords or code lines) with its counterpart in the new environment. If this counterpart is not direct, it will be defined by the researchers.

- Develop a new PC tool based on research made for OO systems, to be tested on the study case. This new tool will be created from the scratch, using assumptions or algorithms already defined in OO based works and addressing directly the new environment and the information it naturally highlights. With this we aim to skip any noise that the original paradigm could add to our solution (like dependency on certain structures or code artifacts) to develop a tool fully independent of the original language.

- With the help of an expert in the study case, evaluate the utility of the developed tools, in terms of usability and helpfulness.

- Discuss the overall results of the evaluations, aiming to determine if the port from OO to non-OO systems maintained the performance of the tools

The rest of the thesis continues as follows: in chapter 2 we will describe the related work done in PC, chapter 3 will describe the methodology to prove the hypothesis and the tools developed, chapter 4 will describe the experiments and its results and chapter 5 will discuss the conclusions about the results, the implications these results have and the future venues of work in the area.

2. RELATED WORK

In this chapter we are going to review what has been done in Program Comprehension (PC) in the last years, both in research and industry. Also we are going to state the main differences between static and dynamic analsys.

2.1 Brief history of Program Comprehension

Programmers always had to understand code. The first formal study about this topic can be attributed to the work of (Brooke R., 1975), where he proposes a model of the cognitive process involved in the code generation behavior of a typical FORTRAN programmer. In this early work he already identifies three phases of the programming process: the understanding, the planning and the coding; and also states that before the programmer can actually start to work on the problem, he has to build representations of the problem's elements and properties, such as initial state, goal, etc.; what composes the "understanding" phase of the problem, not the code.

Since then, there has been a lot of research about this "understanding" phase, which has led to several approaches, theories and conclusions about how programmers understand code. This research can be divided in two main areas: static analysis, which focuses on what kind of information we can get from the source code, documentation, etc.; and dynamic analysis, that focuses in the information we can gather from the application in runtime, like message passing, calls stack, etc.

2.2 Dynamic Analysis for Program Comprehension

As we just stated, dynamic analysis consist on gathering and analyzing information of software generated while it's running. This information can be the program's calls stack, the values a variable takes during the execution, the code artifacts (variables, methods, etc.) invoked to perform the task, and generally speaking, any information you can get during a debugging process, i.e., the information that the computer handles at machine level. Dynamic analysis has the potential to provide an accurate picture of a system because it exposes the system's actual behavior, but it only can provide partial information of the system, since it depends on the scenario executed for the analysis. In the last years, dynamic analysis has gotten a lot of attention, covering a big share of all the research on PC.

A good survey of the "state of the art" can be found in the work of (Cornelissen, Zaidman, van Deursen, et al., 2009), a work that categorizes the last ten years of research in PC, giving an insight of the actual tendencies, possible venues for future work and some discussion about why the research has turned the way it is actually. One of the most relevant observations that this survey does is that most of the articles reviewed focus their research on object-oriented systems. The authors believe that this may be due to the ease of instrumentation, which is a technique that allows extracting information from runtime through the insertion of "inspector functions" in the code that will collect the desired information if the program passes through them, suitability of certain visualizations like UML, or simply the fact that researchers are interested in OO.

2.3 Program Comprehension in the Software Development Industry

It is clear now that PC is a widely researched field, and thus, a lot of algorithms and tools have been developed through the years, but how much of this effort is really being used in the software development industry? There are actually several PC and Software Maintenance tools available, both for OO and non-OO languages, but most of these tools are code reviewers/browsers, which represent a very basic implementation of static analysis. There are several kinds of tools related with Software Maintenance, but that aren't PC tools as such. One example is the project management software, which consists on a suite of tools to create, manage and maintain a complete project. Another example of this is the performance analysis tools, which aids the programmer to test the performance of the application, i.e. the time spent on resolving the task, memory used, etc. Nevertheless, there is a good amount of PC tools, both commercial and academic, that are available for being used in a software development process.

2.3.1 Static Analysis Tools

As we stated in the previous paragraph, the most common tool in static analysis is the code browser, a tool that allows navigating the source code of a project in an easier way that the common text editor. The objective of these tools is to reduce the time a programmer must spend reading code, documentation or any other plain text related to the system he wants to understand.

A good example of this kind of tool is Juliet (infotectonica sa, 2006), a code browser for Java projects that allows you to navigate the code as if it was hypertext, that is, shows you information related to the code when you hover the mouse over or when you drag and drop a code snippet in a special windows, etc.; also, it can give you cross-

information of several code objects while you are browsing it (which functions modify certain variable, which classes use certain functions, etc.).

Another kind of tool that belongs to the static analysis is the metrics generator tool[1], which calculates different metrics of the target system using text analysis techniques like text mining, parsing, etc. The objective of these tools is to give to the programmer a set of values he can use to compare and analyze different code objects in a more quantitative way than basing his analysis only in the code. An example of these tools is Understand (Scientific Toolworks Inc., 2010), a tool that integrates metrics generation with several features, allowing you to generate reports with statistical information about your code, which makes it easier to measure and evaluate.

There is a third kind of static analysis tool that is used inside the industry, the Program Slicing tools. Program Slicing is a technique introduced in 1981 by (Weiser M., 1981) which consist in select a variable/statement to be analyzed, and then iteratively cut code from the program that does not affect this variable/statement until nothing else can be cut without affecting its original value. The resulting new program is called a slice, a program a lot smaller than the original that affects the target value/statement in exactly the same way that the full program, since it has only the code strictly related to that value/statement. The objective of these tools is to reduce the amount of code that a programmer has to read in order to understand certain feature to a minimum, easing the understanding process. Unravel (The Unravel Project, 1999) represents a good example of these approach to static analysis. This tool was meant to help to identify common code amongst several executions of a program.

[1] There are some dynamic analysis tools that also calculate metrics of the trace, but most of the metrics generation was found to be in static analysis tools.

2.3.2 Dynamic Analysis Tools

As we stated in 2.2, dynamic analysis consist in analyzing the information a program generates when it runs. Even if most of the tools available for PC are not dynamic analysis based tools, there are still a good number of tools in the market. The most common kind is the Trace Visualization tool, which shows the information related to an execution trace, i.e. the call tree, in a graphical way, likely easier to understand. The objective of these tools is both to somehow compress the huge amount of information a trace can generate, taking advantage of the ability of humans to interpret a lot of information from pictures, and to give an insight of how a particular execution of the program looks, making easier to a newcomer programmer to understand the big picture of the system at hand. A good example of trace visualization is AVID (University of British Columbia, 2003), a tool that animates the execution trace of a Java program in a way similar to an UML diagram, counting the messages going in and out from each entity at each step. The special feature of this tool is that the entities shown on the diagram can be user-defined, this is, the user may define any subset of the system as an entity, let it be an entire package, a subset of a library, etc.

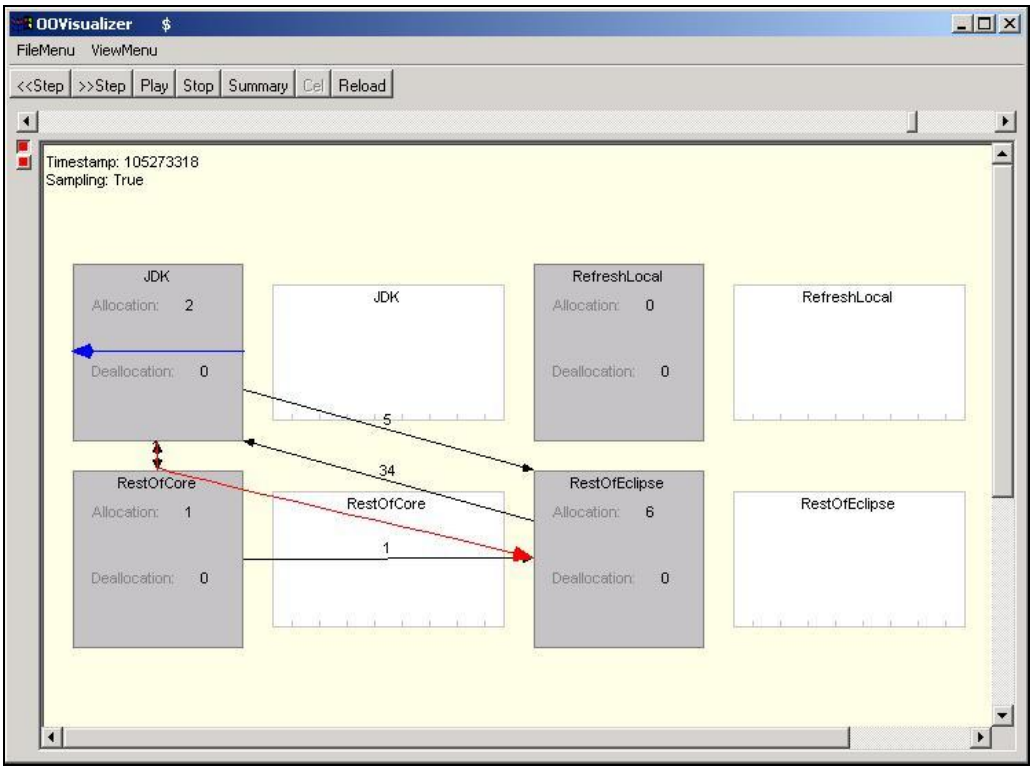


Figure 1: AVID Screenshot

Figure 1 shows a screenshot of AVID visualizing part of the execution of the Eclipse integrated development environment. Boxes represent components, such as the JDK, a collection of core Eclipse components (RestOfCore), the functionality associated with refreshing a local directory (RefreshLocal), and remaining Eclipse (The Eclipse Foundation, 2001) components (RestOfEclipse). These components are defined by the user by describing a name for the component and by describing what packages, classes, methods, etc. in the execution contribute to each component. Black arrows represent the messages that have been sent between components. Colored arrows show the currently executing threads. The number of object allocations and deallocations to the point in the animation are shown in each component.

This tool could be of great help in our case because it will allow the expert to make use of his experience in order to define macro-entities that explain better the behavior of the system, so any analysis based on these entities will be clearer and easier to understand. However, the main issue that we find when trying to use this tool in our case is the fact that it works as a plug-in for the IDE Eclipse (The Eclipse Foundation, 2001), so it's restricted to the languages that tool supports. However, the idea itself of a visualization tool that lets you define the entities you want to see interact may be adapted successfully to any paradigm, as long as it has a recognizable entities structure with message passing between them. An exception to this condition can be the functional languages.

Also, there is another kind of tool that uses dynamic analysis that is present in the industry, the Debugging Tool. Debugging is a classic way to understand software behavior, in which the reviewer analyzes the state of each variable at each statement of the program, searching for the point in which the bug was generated. Actually, there are a few new approaches to this method. One that is worth mentioning is the method used in the tool RetroVue (VisiComp Inc., 2004) that is simply an offline debugger, which first stores the information of the execution, to be analyzed later. This allows the feature of going backwards in a debug process, which is normally impossible in any debugger. Since our study case doesn't have a debugger included, this tool will be of great help, especially with its rewind feature.

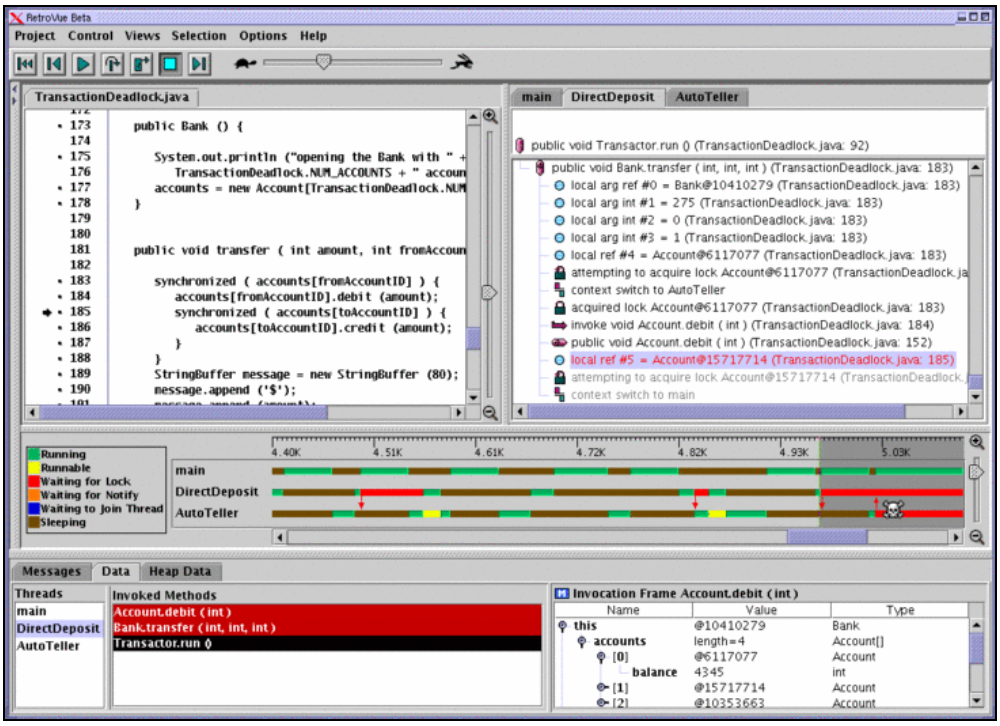


Figure 2: RetroVue Screenshot

Although a debugger doesn't depends on the paradigm of the language it's been applied, RetroVue can't be used in our case first because it's meant for Java and second because, as we will detail later, our study case is a very complex embedded system, and thus we don't have access to the call stack at a processor level, and is impossible to instrumentalize the entire code to know the complete state of the system (variables, calls, etc.) at any moment.

Also in the line of debugging there is another method that innovates in this field, the Automatic Debugging. This method consists on automatically find the cause of a bug using automated testing. The creator of this method is Andreas Zeller (Zeller, 2000), and his tool Delta Debugger (Lehrstuhl für Softwaretechnik - Universität des Saarlandes, 2010) is well known in Software Engineering.

The tool developed by Zeller takes the system code, the user input and an automated test able to prove if the target bug has happened or not in certain execution, and then it automatically modifies either the source code, removing code or adding code from a working release, or the user interactions; until the input can't be modified any more, this is, it has become an irreducible statement or any further modification to the input will not reproduce the bug.

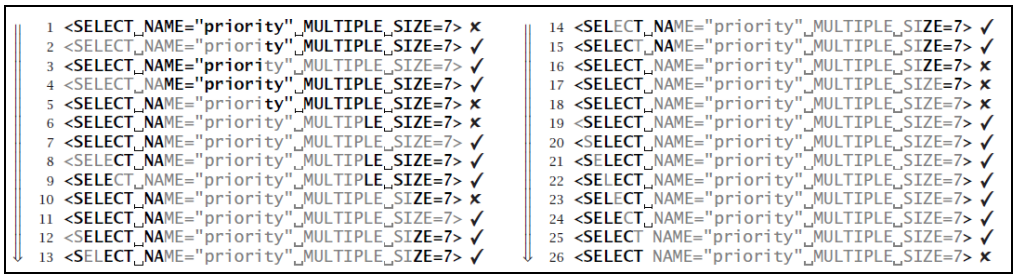


Figure 3: Delta Debugging simplifying a HTML input

Figure 3 shows how the algorithm reduces a HTML input to the smallest possible input that reproduces the bug, in this case a SELECT tag, starting with a SELECT with several parameters but we don't know a priori which of those participate in the bug, so the algorithm automatically removes parts of the input until the bug happens, then that part stays removed and the algorithm continues until nothing else can be removed.

As with the previous tool, the addition of a debugger could be of great help to our study case, since it hasn't anything similar to a debugger, and the automation of the process makes it an even better choice due to the big size of the target system. The problem we found when trying to use this tool is that adding any new feature to the study case is very hard (in fact this work is part of a project that aims to ease the research and programming of new features in the study case), so adding the necessary framework to have automated testing is out of the scope of this work.

Finally, there is also a set of tools that combine both static and dynamic analysis, composing more complete tools that aim to unify all the issues related with PC. From these tools there are two good examples. The first one is Cerberus (Aho et al., 2008), a tool intended for Java programs that uses both information retrieval (static) and elements activation (dynamic) to find concerns as input for a prune method that adds more elements to the concerns previously added. In the first stage, Cerberus automatically extracts a starter set of possible concerns from the source code (method or variable names, comments, etc.) and requirement documents, creating a thesaurus of terms. Then it makes a trace analysis to detect the software elements activated while executed the desired concern, which is combined with the previous thesaurus to create a starter set of program artifacts (classes, methods, etc.) that are likely to perform the desired concern. Finally, a prune algorithm is applied over this starting set, adding or removing program elements according to their relation with the starting set, like heritage, references, containment, etc.

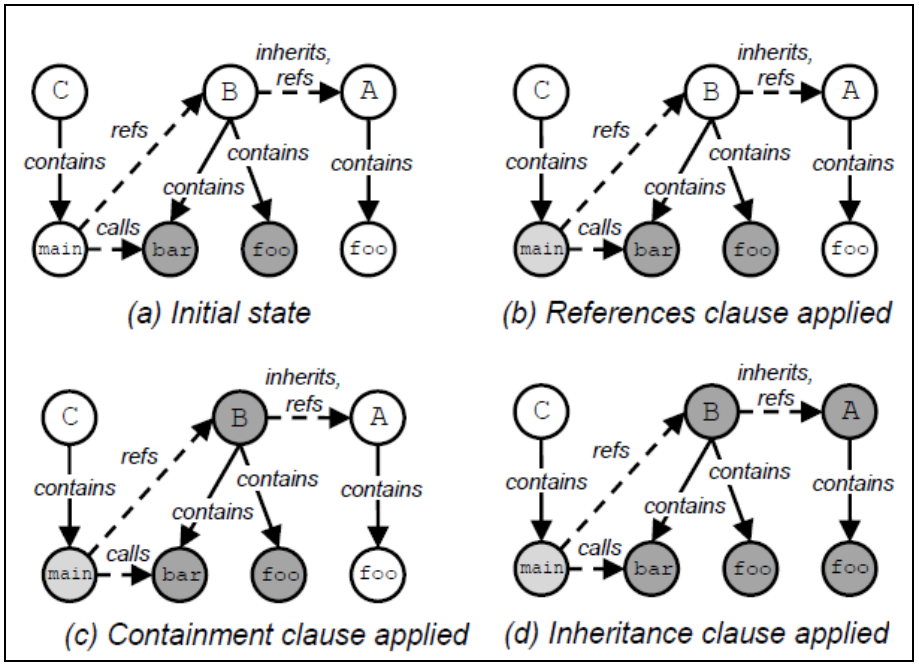


Figure 4: Prune Dependency Analysis

In **Figure 4** we see an example of the rules applied when augmenting the starter concerns set. We start with *bar* and *foo* as the initial elements (a), then we add the *main* method because it directly references *bar* and *foo* (b), then we add the B class because it contains the starting methods (c), and finally we add the A class because it inherits from B (d).

The advantage this tool presents is the ability of automatically finding concerns, which abstracts us of implementation details, allowing a newcomer to locate the artifacts that implements the concerns he wants to review. Nevertheless, this tool can't be applied in our study case because we don't have complete access to the documentation, thus the information retrieval phase will create an incomplete thesaurus, and the later phases will not be as effective as in the original work.

The second one is proposed in (Béron, Henriques, Pereira, & Uzal, 2007) for aid in the understanding of C programs. This tools objective is to help the programmer to navigate the system in the abstraction level he wishes, through several views based on static analysis (assembly and source code views), dynamic analysis (function and modules views), and a fifth behavioral view that aims to relate the functions of the system's code with concepts in the system's design.

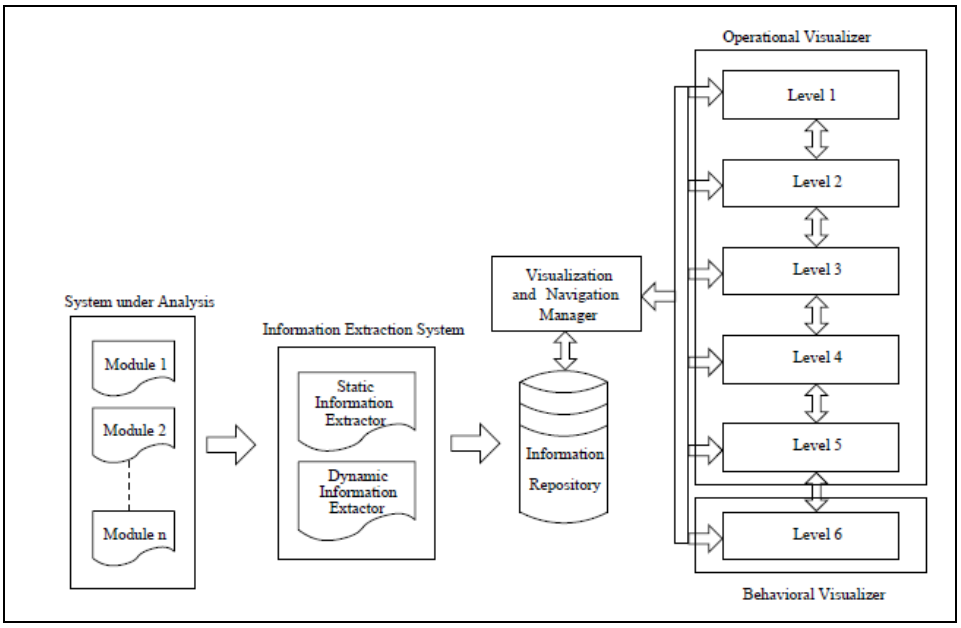


Figure 5: Béron's PC Tool Architecture

As we see in figure 4, this PC tool takes the target system as an input and then, using static and dynamic analysis, it creates an information repository from where the several views will feed. The static analysis recovers the information needed to create the first two levels (assembly and source code), and the dynamic analysis is used to recover the information of the other 3 levels, which are in order a function list for the functions, a function call graph to see the references between functions, a module communication graph to see the relations between system modules. The system output acts as the sixth view.

This tool could represent a great help when trying to understand our study case mainly because, and as will be detailed later on, it can be considered to be implemented with a Domain Specific Languages (DSL, it's a programming language that contains information of the problem domain in its syntax), thus the capability to relate the

elements in the code with elements in the problem's domain can be very useful. Also the fact that it's meant to C programs brings it closer to our problem.

The main issues that prevent us for using this tool in our research are in one hand the fact that we don't have access to the call stack to retrieve the assembly code, and on the other hand the level of instrumentation we can achieve. As we will detail later, our study case is a modification of C in which the extra code comes in the form of commentaries that are pre-compilated and then added to the rest of the C code. This structure makes impossible to determine the points inside of this extra code that we need to instrumentalize in order to retrieve the runtime information related with variables that this tool requires. In our study we made instrumentation in order to retrieve some dynamic information, but the level of instrumentation we needed was much simpler that the one required in (Béron et al., 2007)

From all the tools reviewed in the context of this work, dynamic analysis based tools were the minority and tools meant to be used on object-oriented systems were the majority, which is in touch with the results of (Cornelissen, Zaidman, van Deursen, et al., 2009). Of the few tools designed for non-OO systems, most of them were code browsers, and the only dynamic-based one was a debugger, so we may think that despite the big amount of research in PC, the deployment of these products into the industry is neglected, thus is necessary to evaluate if it is possible to adapt these tools to be used in non-OO environments or it is necessary to develop new tools for this purpose.

3. METHODOLOGY

In this chapter we are going to define the methodology that we will use to prove the hypothesis, we will describe the study case and the tools we will develop.

In order to achieve the objectives stated in section 1.4, we propose the following stages:

- Selection of the study case.
- Selection and adaptation of the first PC tool to be tested.
- Research and development of the second PC tool to be tested.
- Evaluation of the tools supported by the domain expert.
- Discussion.

In this chapter we are going to discuss the first three points

3.1 Selection of the Study Case

The system chosen for this study consists of the software that runs inside the *Magnetic Resonance Imaging* (MRI) scanner *Phillips Intera*. This system called *Gyroscan-NT Pulse Programming* is made in an ad-hoc language called *GOAL-C* (12th Course of the International Zurich Magnetic Resonance Education Center, 2001), which is a modification of C that claims to be OO, but fails at implementing essential features of an OO language, like polymorphism, instantiation of objects, inheritance, etc. The complete project consists of more than 3.259 files, 4.865 functions and 475.167 LOC.

The main goal of this language is to program *Magnetic Resonance Measurements* (MRm), which represent the exams taken with the scanner. These measurements are made of small magnetic resonance sequences or *prepulses* that are already pre-programmed in the machine, so the programmer has to code a set of instructions to run

these prepulses a given amount of times in a given order. To achieve this, programmers must understand several kLOC just to get an insight of how the system works, and then another set of kLOC to know how to program the MRm they wish.

The system's main code is composed of 4 main folders, where each one of them contains hundreds or sometimes thousands of source files, both headers and body files. The file's names are a nomenclature that contains, in the first 2 characters, the folder to which the files belongs and, in the rest of the name, information of the type of prepulse or sequence phase this file is part of. A single file can have several roles, for example, a file can be part of the first validation phase of a diffusion exam, or the main component in the cardiac imaging tests, and so on. This makes hard to find sets of files that implements a concern, because each file can have either a very specific or a very generic role in the system, provoking a concern to be too scattered amongst the files or files that participates in several concerns.

The integration between C code and GOAL-C code also represents an issue when trying to understand this system. Not all the system is written in GOAL-C, but it has some code snippets usually at the beginning of some files, both header files and body files. This code aggregates consist mainly in definitions of global variables, or "objects" according to this system's design, and full definitions of some functions that involve these "objects" (full body definitions, not just the header of the function). These aggregates define features in a way closer to the problem domain rather than the program domain, and thus are very important to understand the complete system.

The problem arises when trying to automatically analyze these parts of the code. For example, we can't instrumentalize them because inserting any normal C code instruction there will make the GOAL-C → C precompiling process fail, and the GOAL-C attachment hasn't any statement related with printing custom information, so we can't know if the program flow passes by there or the state of any variable inside that code,

we can only know when the program uses the “objects” declared there by instrumentalizing the ANSI-C parts of the system.

The only way to interact with this system when you want to develop is through a virtual machine that runs Windows XP Embedded. This virtual machine doesn’t let you install any kind of software, so we can’t use any PC tool we desire, but we can use the developing tool that comes with the machine, Visual Studio. The compiling process is made through a script written in Pearl that first parses the GOAL-C code and then compiles the ANSI-C code. This makes the compiling process a “black box” to the programmer, thus making impossible to interact with this task or modify it in any way.

The attempts to use existing PC tools have failed (Campos, Cortazar, Eterovic, Tejos, & Irarrazaval, 2009), mostly because these tools need either installation or access to the compiling process, both impossible tasks in this environment. This fact encourages the use of PC approaches or algorithms over PC tools, so that was our focus in this work.

Summarizing, the size and complexity of this program makes it a good study case to test the performance of PC tools. However, it has the drawback that it’s difficult to modify the environment in which it runs. Due to the characteristics of the system, we can’t use any commercial or installable PC tool because we can’t install new software, debuggers or tools that need to run side-by-side with the execution because we don’t have access to the compilation process or the execution itself, tools that need detailed metrics about the system because we can’t instrumentalize all the code and specific IDE plug-ins because we can only use the Visual Studio Version that is already installed, without any plug-in. On the other hand, we can still use PC tools that doesn’t need very detailed instrumentalization like trace-analysis tools because we can still instrumentalize the C code, static analysis techniques like pattern detection because we have access to all the source code, metrics analysis tools since we have access to some static and dynamic information, visualization tools since we have a lot of semantic information of the

system embedded in the code itself and code slicing techniques because we can modify the system removing code and execute it to test the output.

3.2 Tools Tested

As we said in the end of section 2.3.2, dynamic analysis tools are not very common in the software industry, but on the other hand, dynamic analysis tools are pretty common in the research. The little penetration that dynamic analysis tools have in the industry may be due to the difficulty of perform a dynamic analysis in such large systems. For example, tools based on scenario analysis may find problems in industrial-size software because it's hard to create enough traces to cover the different use cases possible; trace visualization tools may be unusable due to the size of the traces in this case unless the tool allows some kind of summarizing of traces; etc.

On the other hand, the huge interest in dynamic tools in the research field makes that these tools are far more studied and tested than static tools, thus the works about dynamic analysis should be more reliable.

These two observations led us to think that if the problems derived from the size of the software can be addressed, the use of dynamic tools in industrial software could be very fruitful. The static tools actually present in the industry are too generic, thus the help they can give to the understanding is limited. Besides, the ability of dynamic tools to gather information about use cases could prove to be very useful, because it could help the programmers to address maintenance issues involved with the bugs reported by the users, improving the impression of the client towards the product.

These particular observations led us to try our hypothesis using dynamic analysis tools, since they seem to have a potential to help in industrial software that has not been

exploited yet. If dynamic analysis tools were viable in real-size software, the need of porting the research on this field to the industry would arise.

3.2.1 HITS + fe-Tree

One of common approaches used to help the understanding of code, as stated in (Cornelissen, Zaidman, van Deursen, et al., 2009), is feature analysis (Antoniol & Guéhéneuc, 2005). This technique aims to identify subsets of code that are relevant for a certain functionality, which is done typically analyzing execution traces. The goal of this identification is to give the reviewer good starting points to start reading code, because if he knows where are implemented the functionalities he needs to understand, he will take less time reviewing the system because he can skip the useless code.

In order to prove how features analysis works in a non-OO system we tested the HITS algorithm proposed in (Zaidman, Calders, Demeyer, & Paredaens, 2005), a technique originally designed to rank Internet sites. This algorithm works under the typical assumption of features analysis tools, which states that a good starting point to review the code will help to get an insight of the system behavior, making easier the understanding task.

HITS counts the incoming and outgoing calls for each one of the code artifacts, and uses these values to define two variables: one represents the quality of the entity as an *authority* (depending on its incoming calls), and other represents the quality of the entity as a *hub* (depending on its outgoing calls).

The authors assume that a high value of the relation outgoing/incoming calls in an entity means that the class's main function is to delegate responsibilities, thus is coordinating the flow of the execution. On the other hand, a low value of outgoing/incoming calls

suggests that the artifact is being used by other artifacts as a utility (for example *printf* or *System.in*), and thus is not very important to understand the main system's behavior.

$$h_i = \sum_{i \to j} a_j \qquad (1)$$

$$a_j = \sum_{i \to j} h_i \qquad (2)$$

Figure 6: Formulas to calculate Hubiness and Authority

These formulas show that the hubiness depends on the sum of the authorities of the node's children, and the authority depends on the sum of the hubiness of the node's parents.

The *hubiness* (1) depends on the sum of the *authorities* of the child nodes, and the *authority* (2) depends on the sum of the *hubiness* of the father nodes. These values are iteratively updated and normalized until there is convergence. In the next figure, 2 and 3 will be good authorities, and 4 and 5 will be good hubs. The authority of 2 will be bigger than 3, because they share 4 and 5 as fathers, but 1 is a better hub than 2 because it has more children.

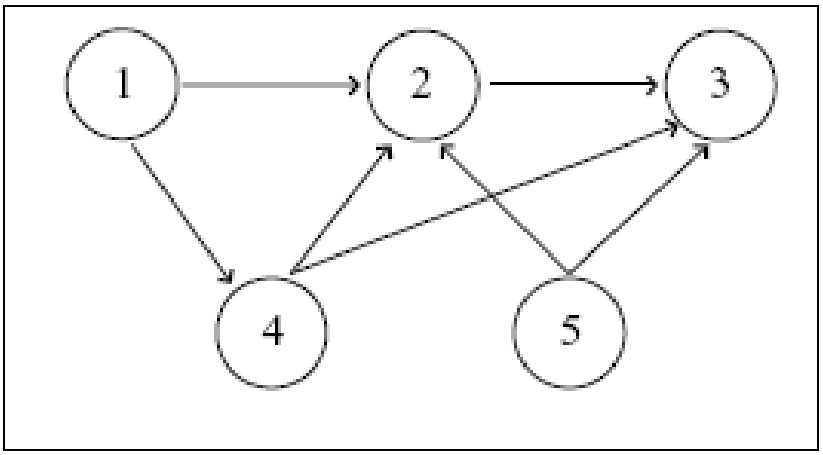


Figure 7: Graph for HITS example

In this graph we see a possible call stack of a program, where the function 1 calls the functions 2 and 4, function 2 calls function 3, and so on.

In Table 1 we see an example of an application of the HITS algorithm on the graph from Figure 6. In a run of the HITS algorithm the hubiness (H) and authority (A) of all the nodes start with value 1, then the formulas (1) and (2) are applied over all the nodes, so after the first iteration the hubiness for all the nodes becomes the count of its children, and the authority the count of its parents. Applying (1) and (2) over these new values will create a clear difference between the nodes with highest H and the rest, the same with the nodes with highest A. in the iteration 3 we can see this difference widens up even more, showing clearly that the nodes 4 and 5 have the highest H, and that nodes 2 and 3 have the highest A.

If we interpret these results from the author's view, the nodes 4 and 5 have a coordinating role in this system, because they are in charge of the main part of all the references, and this makes them good starting point to review the system and understand its flow. In the same line, we can see that the nodes 2 and 3 are the entities that perform most of the work in the system, since they focus the majority of the calls, making them not a good starting point for a newcomer programmer to review because they are likely to have too much implementation details that distract from the system design.

The output of HITS is a ranking of classes according to their hubiness, in which those in the highest positions have better references to authority classes and thus, are a better starting point to review code. In this example, the ranking generated by the algorithm according to the hubiness of these nodes will be nodes 4 and 5 sharing the first place, then node 1 as third place, node 2 as fourth and node 3 as fifth.

Node	**A_0**	**H_0**	**A_1**	**H_1**	**A_2**	**H_2**	**A_3**	**H_3**
1	1	1	0	2	0	4	0	8
2	1	1	3	1	6	3	16	5
3	1	1	3	0	5	0	15	0
4	1	1	1	2	2	6	4	11
5	1	1	0	2	0	6	0	11

Table 3: Example of the application of HITS algorithm

This table shows the different values of hubiness (H_i) and authority (A_i) that the nodes if figure 7 will get in each iteration of HITS

HITS was not necessarily intended to be used only in OO systems, since it has very weak language-related assumptions, i.e. it only assumes a system divided in classes, so the porting to other environments should be just as hard as finding an equivalent division. The problem arises with the validation of the results in new environments. Specifically, OO languages highlights domain information thanks to their classes notation (Wiedenbeck S., 1999), so it highly possible that the information collected by HITS is related with the domain structure of the system. This implies that the equivalent we may find in other environments could remark something different, and then if the quality of the recommendation made by HITS in the original case is related to the role the entity had in the problem domain, changing the paradigm will drop the performance

of the tool because the replacing entity selected may not have nothing to do with the problem domain.

In our study case, the language doesn't achieve any of the characteristics an OO language should have (Booch G., 1986), so we can't say that the new entity selected will highlight domain information like the classes in OO. For this reason we believe that the success of HITS in this new environment will depend on the quality of the target system's design and our ability to choose an entity able to describe the flow of the execution in the highest abstraction level possible.

Since HITS was designed to work on systems divided in classes, we had to choose a new way to separate the software. Together with a domain expert, we decided to try two different ways: dividing the software in functions and dividing it in source files. We believe both partitions can work because our study case is designed such that every function/file has a specific main responsibility (just like a classes system). With this, the output of our modification will be a ranking of functions/files which will work the same way as the class ranking of HITS.

Our alternative finally consisted on using the functions of the study case as the entities of the original algorithm, and the incoming/outgoing calls between two functions as the references between the entities that the algorithm needs. This way, we can calculate the *hubiness/authority* for each function the same way HITS does it in a class-based system, and the output of the algorithm will be a ranking of functions.

Besides this partition, we divided the software using the source files as the entities, and the final *hubiness/authority* value of each file was calculated as the sum of those values for each function inside that file. The motive to do this is to extract all the information possible from the instrumentation.

Our implementation of HITS uses as an input a file created through instrumentation of the target code that lists all the entry and exit points of all the functions involved in certain execution. The file contains information about the name of the function, in which source file it is contained and if it is an incoming call or an outgoing call. Then we extract this information and create a tree to represent the execution flow, where each node represents a function and function B is child of function A if A calls B. To prevent cycles due to a single function being called for different other functions we create a new node in the tree that represents the new appearance of the function in the trace.

When this tree is completed we traverse it counting all the incoming/outgoing calls for each function, so we have the values of the first iteration of the HITS algorithm (amount of children/fathers for each entity). With these values we iterate over all the functions updating their hubiness and authority according to the formulas in figure 5 and then normalizing these values by the highest found in the last iteration until there is convergence or a certain number of iterations are met.

Once the calculations of the hubiness are done, for each source file we look at which functions are defined inside that file and then we use the sum of the hubiness of these functions as the hubiness of this file. With these values we proceed to create a spreadsheet that contains an ordered list of functions and their respective values of hubiness, authority, incoming calls, outgoing calls and Coupling Between Objects (CBO) that measures the references a class holds to other classes (incoming references per outgoing references) (Chidamber & Kemerer, 2002). We will compare the performance between these different measures for the values obtained from the files and the functions.

Since HITS uses information about the execution trace to work and generates a ranking of functions, we developed a more complete tool based on prior research related with these two facts. In (Campos et al., 2009) two tools to understand the Gyroscan code

were proposed, a static analyzer and a dynamic analyzer. The dynamic analyzer implemented a tree visualization similar to the *fe-Tree* proposed in (Béron et al., 2007), which uses code instrumentation to isolate the entry and exit points of each function executed in a particular scenario. We used this tool to retrieve and show the information related to the execution of the program.

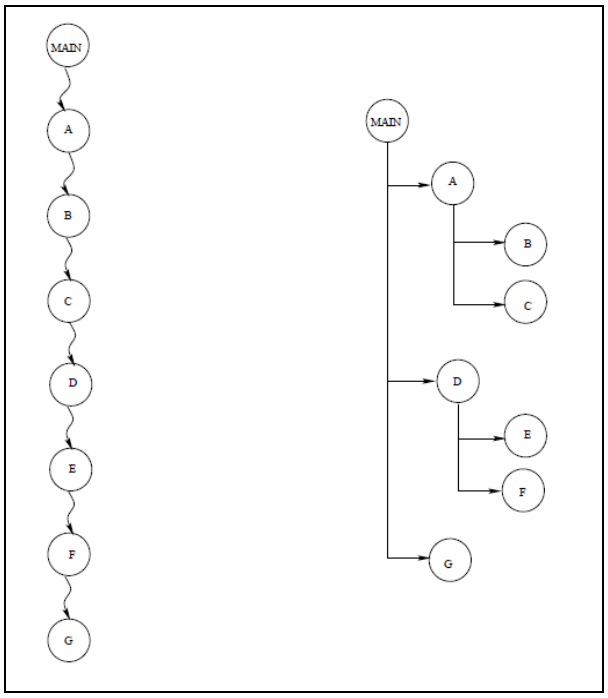


Figure 8: Function list VS fe-Tree

On the left we can see a common call stack, and on the right we see the same call stack ordered according to which function called which other function.

As we see in **Figure 8**, representing a function stack call as a fe-Tree (right) is much clearer both in visualization and dependency than a simple list (left). For the analysis that was done by the algorithm and the end user is more useful to know the direct dependency between functions instead of the whole call stack. For this reason we use this model both in the final visualization of the tool and in the internal model used to analyze the trace.

On the other hand, (Antoniol & Guéhéneuc, 2005) proposes the idea of micro-architecture, or simplified versions of the system's entities, designed to show the programmer only the pieces or "slices" of the execution trace relevant for certain feature. We used the ranking of functions generated by HITS in order to determine which functions should be showed and which should be pruned from the *fe-Tree*.

Our final tool uses in a first stage the trace information collected with code instrumentation to create and show the *fe-Tree* that represents the information and define the initial values of input/output calls of each function, needed for the HITS algorithm. After running HITS on these initial values, the resulting ranking is reported in spreadsheets to be analyzed for the programmer afterward, and also is used to assess which functions should be pruned from the visualization or not. Finally, the initial *fe-Tree* visualization of the complete trace is pruned to show only the upper ranked functions, but preserving their dependency relations, i.e. if function A is pruned, all the children of A pass to be children of A's father. With this, we get a new *fe-Tree* that is easier to analyze, both because it's smaller than the original and because shows only the most relevant functions for the executed feature.

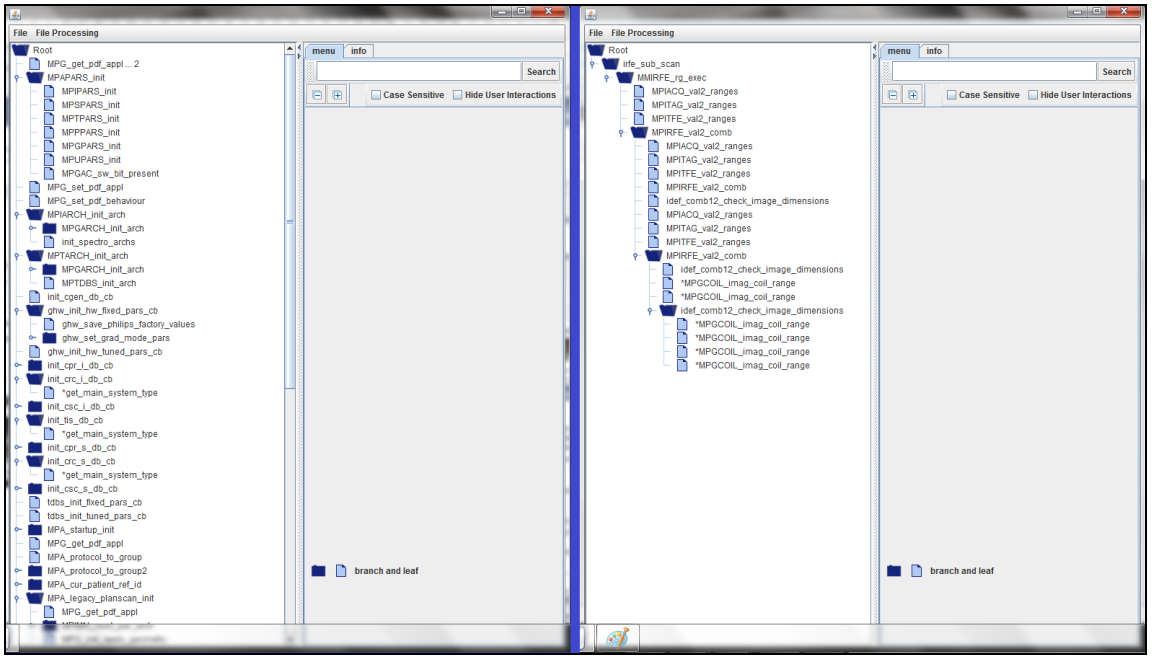


Figure 9: An example of the tool proposed by Campos

On the left we see the complete fe-Tree of the selected execution trace, and on the right we see the resumed trace after pruning the less important functions.

In figure 9 we see two stages of the tool proposed by (Campos et al., 2009). In the left side of the blue bar we see the main screen of the tool after loading an input file with 193368 lines, i.e. an execution trace that involved 96684 function calls. In the right side we see the tool after applying the HITS algorithm over the trace. The ranking created by HITS is used then to select only the 80% most relevant and show it to the user, that as we can see consist of only of 23 function calls. This means that the functions involved in these 23 calls are in charge of the main flow of tis execution.

3.2.2 SORter

Another common approach to PC is software visualization, which consists in the process of modeling software systems for comprehension of relevant features of the system (Price, Baecker, & Small, 1993).

To analyze the performance of visualization tools, we evaluated in a first stage of this work the possibility of using Alma^2 (Alma Two) (Oliveira N., 2010), a visualization tool meant for DSL, which works under the assumption that since in a DSL the problem and program domain are very close to each other, it's vital for understanding that both domains can be mapped to the source code and visualized in sync. To achieve this, the tool displays an animation based on the semantic information contained in the language, along with the source code and the grammar's derivation tree, in order to help the reviewer to relate the program and problem domain.

The advantage of using this tool is the independence of the target language as long as it meets the basic requirements of a DSL. In figure 8 we can see a screenshot of Alma^2. We can get a lot of useful information about the system through the "debugger" on the left side (a variable watcher and a line-to-line code viewer), and the problem it is trying to answer through the animation and the call stack on the right side.

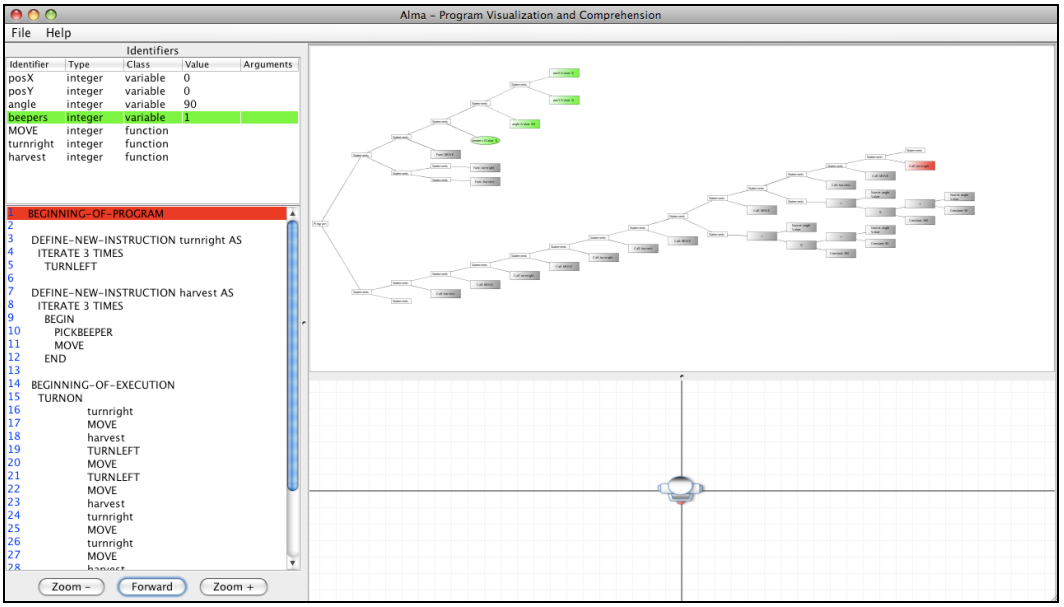


Figure 10: Alma^2 Views

Although this tool's main feature was its capacity to be used in any system despite the language, this same generality made the tool somehow "stiff"; for example, the animations could be just still images being moved around the window, the grammar's derivation tree could be only zoomed in/out, etc. For this reason, we took the decision to develop our own visualization tool inspired in the philosophy behind Alma^2 and based on two visualization approaches widely researched.

The tool we developed is not an animation suite as Alma^2, instead it is a visualizer more like Alma^2's derivation tree, but it preserves the feature of independency from the target language. Also it addresses the same problem that Alma^2, it helps the programmer to relate the problem and program domain, showing him the interactions between objects that model the real-life problem.

In detail, SORter is a dynamic-based visualization tool designed to help understanding the Gyroscan code or any other system meant to program sequences of actions (for

example the programming of a production line). The objective of this tool is to help the reviewer to understand the execution flow of the actions involved at a problem level, showing the information related to a particular execution using language and concepts from the problem domain. To achieve this, SORter displays the sequences of actions in a hierarchical view, helpful to understand dependency between actions, and also in a temporal view, helpful to understand the relations between the durations and the starting times of those actions. The big difference between SORter and any process modeling tool is that the latter does not take necessarily into account the starting/ending time or duration of each process.

The tool's input is a hierarchical outline, i.e. a text representation of a nested or hierarchical structure, in which the "nodes" represent the name of the action executed and the "leaves" represent the available information about the action named in those leaves' parent. The idea behind the tree representation is to provide different levels of abstraction, i.e. the root represents the whole process, later nodes represent more detailed definitions of the actions involved, and the nodes in the second-last level represent the actions themselves.

```
SORterInput=============================================================
SOR`kernel:ref_obj = [ SOR`cycle (0)     ], dur = EMPTY
SOR`kernel:dur     = 1542.8572
SOR`kernel:ref     = 578.6880
SOR`kernel:dur2    = 964.1692
SOR`kernel:min_dur = 1542.8572
SOR`kernel:sar     = 11420.5918
SOR`kernel:rf_tot_gating_dur  = 114.8064
SOR`kernel:rf_max_b1= 25.7249
#1 *SOR`cycle       (rf enabled)
#1  SOR`cycle       (rf enabled)
    SOR`cycle:ref_obj = [ SOR`phase (0)     ], dur = EMPTY
    SOR`cycle:dur     = 771.4286
    SOR`cycle:ref     = 578.6880
    SOR`cycle:dur2    = 192.7406
    SOR`cycle:min_dur = 771.4286
    SOR`cycle:sar     = 5710.2959
    SOR`cycle:rf_tot_gating_dur  = 57.4032
    SOR`cycle:rf_max_b1= 25.7249
    #1  SOR`tnt         (rf enabled)
        SOR`tnt:ref_obj = [ undefined (0)     ], dur = EMPTY
        SOR`tnt:dur     = 0.0000
        SOR`tnt:ref     = 0.0000
        SOR`tnt:dur2    = 0.0000
        SOR`tnt:min_dur = 0.0000
        SOR`tnt:sar     = 0.0000
        SOR`tnt:rf_tot_gating_dur  = 0.0000
        SOR`tnt:rf_max_b1= 0.0000
    #1  SOR`bbi         (rf enabled)
        SOR`bbi:ref_obj = [ undefined (0)     ], dur = EMPTY
        SOR`bbi:dur     = 525.3918
        SOR`bbi:ref     = 0.0000
        SOR`bbi:dur2    = 525.3918
        SOR`bbi:min_dur = 525.3918
        SOR`bbi:sar     = 1039.5174
        SOR`bbi:rf_tot_gating_dur  = 23.8728
        SOR`bbi:rf_max_b1= 15.7127
```

Figure 11: Example of SORter input

Figure 11 *shows a fragment of the hierarchical outline that the Gyroscan produces after a scan is performed.*

After the input was read, SORter displays the hierarchical outline in a typical tree visualization, similar to the hierarchical graphs of Understand (Scientific Toolworks Inc., 2010), but with visual captions as suggested in (Shilling, Stasko, Graphics, & Center, 1992). In our implementation the colors of the nodes symbolize the different kinds of nodes that are present in the sequence, which can be user defined. When you right-click a node, a small window with the information of that node pops out.

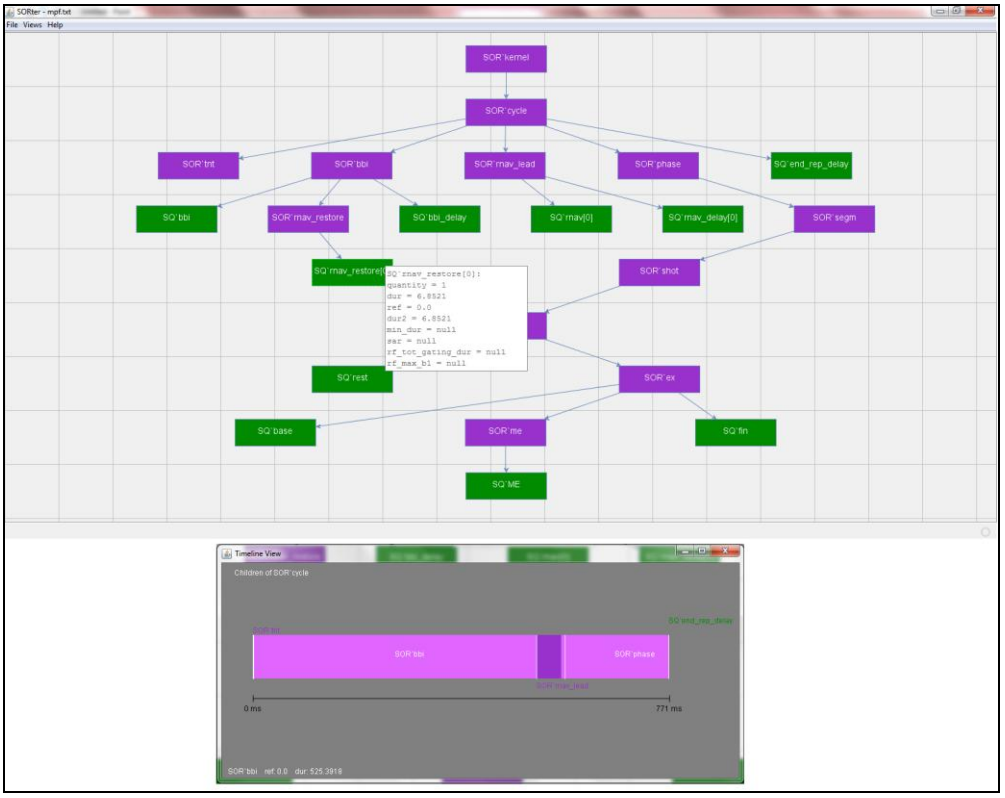


Figure 12: SORter Views

The gray window in the corner shows the timeline view of the tool, and in the background we can see the main tree view of the tool.

Also SORter has a temporal view of the different levels of the tree, the children of a particular node and all the leaves of the tree, ordered by their starting time and where the size of the rectangles gives a reference of the duration of each action. This view is a modified version of the temporal view proposed in (Bohnet, Voigt, & Doellner, 2008). The original view showed in the x-axis the time of the execution and in the y-axis the functions being active in that moment of time. In our representation, the x-axis also shows the time of the execution, but instead of having a y-axis showing the hierarchical levels in our case, we choose to show a determined set of nodes at a time to make the visualization clearer.

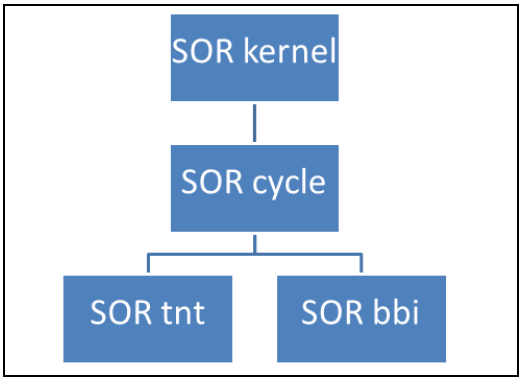


Figure 13: SORter tree view for the example input

This figure is the tree representation of the hierarchical outline of ***Figure 11***.

If we take the figure 8 as an input example, we can see that the information showed there has a SOR kernel object as a root, then a SOR cycle object as a child, and this cycle has a SOR tnt object and a SOR bbi object as children. Since these 4 objects are from SOR type, they will all have nodes with the same color. Also if we look at the leafs we can see that the SOR tnt comes first with a duration of 0 seconds, and then SOR bbi with a duration of 525 seconds, so in the temporal view SOR tnt will have a very little space in the diagram and SOR bbi will use most of the room.

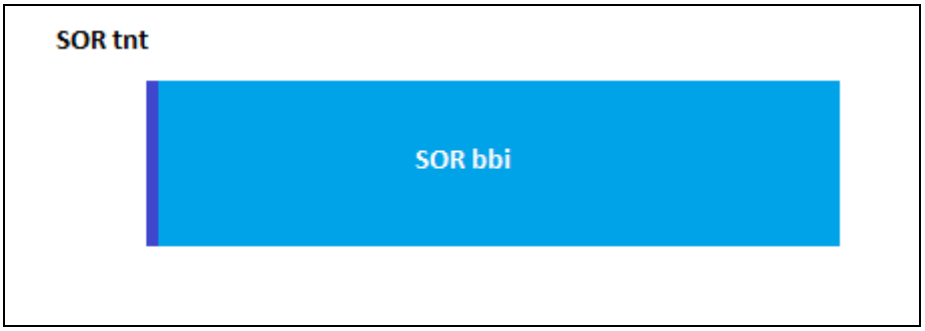


Figure 14: SORter temporal view for the input example

This figure shows a representation in scale of the time span the SOR bbi and SOR tnt have inside the SOR cycle prepulse in ***Figure 11****.*

The information that SORter uses comes directly from a log file generated by the MRI scanner when an exam is taken. This data is also used by an add-on of the system called Sequence Development Mode, which shows the information in plain text, unlike SORter that shows it in a graphical way. We will use this add-on as a reference when evaluating the performance of SORter.

4. EXPERIMENTS AND RESULTS

In this chapter we are going to explain the experimental design to test our two tools, and then we will describe and analyze the results of each one of them.

4.1 Experiment Design

In section 1.3 we defined the hypothesis of our work, which states that PC tools developed for OO systems can't be directly applied to non-OO systems due to the paradigm change, but the research and assumptions that support those tools can be used to create new tools that are less dependent on the paradigm of the target language. In order to prove our hypothesis we had our tools evaluated by an expert sequence programmer, who happens to be also an actual end user of the Gyroscan system. The objective of the evaluations was to validate the usability and helpfulness of the tool, and then to compare these results with those that the tools obtained in their respective original researches.

The first task is the evaluation of an adaptation of the HITS algorithm proposed in (Zaidman et al., 2005), the HITS + fe-Tree tool. For this we will use a similar evaluation to the one performed in original work, which consisted in a comparison between the most important entities' ranking generated by HITS and a most important entities' ranking according to the documentation of the study case.

As we stated in section 3.2.1, the algorithm's ranking is created by ordering the entities by their *hubiness*, but we also looked at a few different static measures in order to compare HITS accuracy with these different measure's accuracy. On the other hand, and since the documentation of Gyroscan is too broad to help to create the desired entities ranking, an expert's ranking was created by ordering the entities by a score that the

expert himself assigned to each entity: from 0, meaning the entity doesn't need to be read to understand this particular execution, to 4, meaning the entity is highly important to understand the execution of the system. By comparing both rankings we can measure the accuracy the algorithm had in this new environment, and thus we can tell if the paradigm shift had any effect on the performance of the tool.

In order to generate the different rankings, all the accessible code was instrumentalized with output sentences to know when the control flow entered or exited a determined function or file. With this we got the input that HITS needs, a list with all the function calls, that allows us to create the fe-tree and let us count the incoming and outgoing references that we will use later to calculate the different measures. All of these measures can be calculated directly with the incoming and/or outgoing references except the hubiness and authority that, as we saw in 3.2.1, need to iteratively update the values until there is convergence. As in (Zaidman et al., 2005), we used a threshold of 0,5 to determine the algorithm has converged.

The second task is the evaluation of SORter, our visualization tool. For this, we will make a controlled experiment similar to the one done in (Cornelissen, Zaidman, Van Rompaey, & van Deursen, 2009). In our experiment, two groups of engineering students will answer a questionnaire, where their understanding of the structure of a resonance sequence was tested: one group will have the help of SORter, and the other the help of Sequence Development Mode (SeqDev). Before the tests were performed, all of the students had a brief training in the use of their respective tool. With this experiment we want to see if there are any differences, either in correctness or speed of the answers, between the users of SORter and the users of SeqDev.

The study group is composed by six electric engineering students. They were part of a sequence programming course, where they learnt the basics of programming the MRI scanner, which assures a balanced level of programming skills in all of them. We are

aware that 6 samples are not enough to be statistically significant, but these results can give us an insight of how different kinds of tools can have a different impact on the user performance.

The tests were held individually to prevent the interaction between participants. There will not be time limit to reduce the pressure, but the times of each task were measured. To have an accurate measure of the time, once a task is completed, it will be impossible to re-take it later on. All the questions are of open answer, and all have a score that ranges from 0 to 4 depending on the completeness, where 0 is the lowest score and 4 is the highest.

The test was composed of an introduction where the motivation and the evaluation were explained, 3 questions that evaluated different abilities and a last question to gather the impressions of the tool used. All of the questions were based on sequences examples chosen by the domain expert according to each task.

The first task was to show a general understanding of the basic structure of a sequence by the explanation of the parts of an example. The second task was a comparison between two examples, meant to test the detailed understanding of the parts of a prepulse. The third task consisted in identifying inconsistencies between the parameters used in a prepulse and the output of the system, meant for testing the ability to find errors in an execution. The fourth question was simply to recover the impressions of the user, to let us to determine if the tool he used was of real help.

Regarding the evaluation of the tests, the domain expert scored each question according to the scale mentioned in earlier in this same chapter. The impressions collected in the fourth question helped to get a more deep insight of the real pros and cons of the tools.

4.2 Experiment's Results

The main result of this thesis is a field study of the portability of PC tools and approaches from OO environments to non-OO environments. This study analyzes the performance of the tools/approaches in the new environment and compares it with the performance in the original environment. This aims to shorten the gap between the research and the industry of PC, showing the relevance of researching for the industry necessities and not only for the academic value.

Another product of this work is the development of two PC tools that will help the research and development of new resonance sequences in the Biomedical Imaging Center (CIB) of the Pontificia Universidad Catolica de Chile.

4.2.1 HITS + fe-Tree

The HITS algorithm was applied to the execution of a typical heart scan, which passed by 6220 functions spread in 202 source files. We used as termination criteria for the algorithm the convergence of the values after being normalized, with a threshold of 0,5. As in (Zaidman et al., 2005), we looked also at some other measures in order to validate the superiority of dynamic measures over the others. Besides hubiness and authority, for each function and file we determined their incoming calls amount (fan-in), outgoing calls amount (fan-out), and coupling between objects (CBO) (Chidamber & Kemerer, 2002). All these measures represent good examples of static and dynamic measures.

In a first stage, the expert determined that a ranking of functions will be of little or no help to a sequence programmer, due to the big amount of functions called and the fact that in this system several functions can be called and not realize any real job (just ask a condition that if is not meet, the function exits immediately). This led us to work only

with the files ranking that, according to the expert, could be very useful to an amateur sequence programmer, since the files involved in a typical maintenance task are very few.

The files ranking the expert created had less than 30 values different from zero, i.e. less than 30 files that are worth reading on a comprehension task. Although this shows the system is well enough designed, this fact prevents us from using a classifier to find a combination of measures (CBO, hubiness, etc.) that could explain the expert's ranking (no classifier can be trained with less than 30 values). With this in mind, we will make the analysis focusing on validate if the hubiness matches the expert's ranking better than all the other measures, as in (Zaidman et al., 2005).

4.2.2 SORter

The usability test ran on SORter was designed so the subjects had to use similar skills to those needed in a typical MRI programming task, as in the experiment done in (Cornelissen, Zaidman, Van Rompaey, et al., 2009), but due to that we had very few study subjects available, the amount of samples was smaller. The test consisted in a questionnaire with 4 questions. The first one aimed to measure the general MRI understanding of the participants, the second one intended to measure their real knowledge about the meaning of the different prepulses involved in a typical exam, the third question stressed their abilities to find semantic errors in the output and the fourth question was intended to collect their impressions of the tool they used.

Six students took the test, 3 with the help of SORter and 3 with the help of SeqDev, all of them in the same computer to discard any noise in the time measured due to computer performance issues. The time of each task was taken with a chronometer and a brief description of the test was given by the examiner before starting. As we stated in 4.1, the

scoring of each answer depended from the completeness of it, specifically of the student's ability to recognize and describe the function of each prepulse present in the sequence.

4.3 Data Analysis

4.3.1 HITS + fe-Tree

From the 202 files reviewed both by the algorithm and the expert, just 21 proven to be worth reading by the later, so in order to evaluate HITS' accuracy we compared these 21 files with the 30 higher-ranked files by each one of the measures.

Fan-in	Fan-out	Authority
mmiffe__mxg.c(1)	mmiffe__mxg.c(1)	mmiffe__mxg.c(1)
mmirnav__mxg.c(3)	mpidef__g.c(2)	mmirfe__mxg.c(17)
mmirfe__mxg.c(17)	mpimn__g.c	mpidef_comb__g.c(6)
mmipda__mxg.c	mmipr__mxg.c	mpigeo__g.c
mpiffe_sq__g.c(4)	mmirnav__mxg.c(3)	mpicsc__g.c
mmicard__mxg.c(8)	mpicoil__g.c	mpiffe__g.c(5)
mpigeo__g.c	mpiffe_sq__g.c(4)	mpicard__g.c(12)
mpidef_comb__g.c(6)	mpipr__g.c	mpiacq__g.c(11)
mmitrack__mxg.c	mpigeo__g.c	mpgcoil__g.c
mpiffe__g.c(5)	mmppo__mxg.c	mmirnav__mxg.c(3)
mpicsc__g.c	mpaldm__g.c	mpirfe__g.c
mpicard__g.c(12)	mpima0__g.c	mpidyn__g.c
mmipr__mxg.c	mpiffe__g.c(5)	mpidiff__g.c
mpiacq__g.c(11)	mpgcoil__g.c	mpspydec_val__g.c
mpidef__g.c(2)	mpirc__g.c	mpiproc__g.c
mpicoil__g.c	mptspk__g.c	mpicoil__g.c
mpirfe__g.c	mpirest__g.c(15)	mpa_legacy__g.c
mpgcoil__g.c	mpirnav__g.c(10)	mpitfe__g.c(7)
mpidiff__g.c	mmimadec__mxg.c	mpirnav__g.c(10)
mpidyn__g.c	mmpf0__mxg.c	mpscoil__g.c
mpspydec_val__g.c	mpidef_comb__g.c(6)	mpitfepp__g.c
mpiproc__g.c	mpiusegeom__g.c	mpaldm__g.c
mpirest__g.c(15)	mpiacq__g.c(11)	mpitag__g.c
mmg__mxg.c	mpa_legacy__g.c	mpiresp__g.c
mpirnav__g.c(10)	mmpas__mxg.c	mpirest__g.c(15)
mpg2dp__g.c	mpimp__g.c	mpispir__g.c(13)
mpitfe__g.c(7)	mpghw__g.c	mpipc__g.c
mpiusegeom__g.c	mpidef_vol__g.c(16)	mpilolo__g.c
mpispir__g.c(13)	mpicard__g.c(12)	mpscrc__g.c
mpipr__g.c	mpispir__g.c(13)	mpa__g.c

Hubiness	CBO	Expert's score
mmiffe__mxg.c(1)	mmiffe__mxg.c(1)	mmiffe__mxg.c
mpgcoil__g.c	mpidef__g.c(2)	mpidef__g.c
mpicoil__g.c	mpimn__g.c	mmirnav__mxg.c
mptspk__g.c	mmipr__mxg.c	mpiffe_sq__g.c
mpidef__g.c(2)	mmirnav__mxg.c(3)	mpiffe__g.c
mpaldm__g.c	mpicoil__g.c	mpidef_comb__g.c
mpa_legacy__g.c	mpiffe_sq__g.c(4)	mpitfe__g.c
mpatable__g.c	mpipr__g.c	mmicard__mxg.c
mpidef_comb__g.c(6)	mpigeo__g.c	mpidef_enc__g.c
mpiffe_sq__g.c(4)	mmppo__mxg.c	mpirnav__g.c
mpa__g.c	mpima0__g.c	mpiacq__g.c
mpitfe__g.c(7)	mpgcoil__g.c	mpicard__g.c
mpitag__g.c	mpiffe__g.c(5)	mpispir__g.c
mpiacq__g.c(11)	mpaldm__g.c	mpit2prep__g.c
mpiproc__g.c	mpirc__g.c	mpirest__g.c
mpidiff__g.c	mptspk__g.c	mpidef_vol__g.c
mpidyn__g.c	mpirest__g.c(15)	mmirfe__mxg.c
mpiffe__g.c(5)	mpirnav__g.c(10)	mpidef_lord__g.c
mpirc__g.c	mmimadec__mxg.c	mpi2dssp__g.c
mmipr__mxg.c	mmpf0__mxg.c	mmiia__mxg.c
mmirnav__mxg.c(3)	mpidef_comb__g.c(6)	mmirest__mxg.c
mpimn__g.c	mpiusegeom__g.c	
mpirnav__g.c(10)	mpiacq__g.c(11)	
mpima0__g.c	mpimp__g.c	
mpaexec__g.c	mmpas__mxg.c	
mpimp__g.c	mpghw__g.c	
mpicard__g.c(12)	mpicard__g.c(12)	
mpiresp__g.c	mpidef_vol__g.c(16)	
mpimtc__g.c	mpispir__g.c(13)	
mpicsc__g.c	mpidef_grad__g.c	

Table 4 - Thirty top ranked files and expert's 21 recommended files.

Here we see the different rankings that the different measures produced compared against the expert's opinion.

In the previous table are listed the highest ranked files by the different measures along with the expert's 21 recommended files. The colors of the cells are just a visual help that supports the numbers between parentheses, which represent the position of the file in the

expert's list. These colors are meant to aid finding the expert list's files in the other rankings, where greener colors are for the highest ranked and redder are for the lowest ranked.

The 5 measures ranked as first priority one of the two most important files according to the expert, but none of them were able to find all the 21 files. CBO missed 9 files, hubiness missed 11, authority missed 10, fan-out missed 9 and fan-in missed 7. There were 6 files that weren't found by any measure, corresponding to the ninth, fourteenth and the last 4 in the expert's ranking, with a priority of 3 to the ninth, 2 to the fourteenth and 1 to the rest according to the 0 to 4 scale used to rank the files.

The expert claimed that the first ranked file by all the measures is a "must read" if you are trying to understand the heart scan we used to generate the data, so we can expect that the 5 measures are giving useful information. Of the missing files from the experts list, 4 are the last files from the list and the other 2 are rather near the half of the list, so we can say that the different measures prioritized the higher ranked files.

Speaking in terms of false positives and false negatives, the hubiness was actually the worst measure, with 52% false negatives and 66% false positives. The best measure was the fan-in amount with 33% false negatives and 53% false positives, followed by the CBO with 42% false negatives and 60% false positives. In (Zaidman et al., 2005), the hubiness had 10% false negatives and 40% false positives, against 50% false negatives and 60% false positives of the CBO.

One of the issues involved in the difference in accuracy of the hubiness between this work and (Zaidman et al., 2005) may be the subjectivity of the expert's opinion against the objectivity of the documentation used in the original work of HITS. We must remember that for this work we hadn't the help of neither one of the developers of the

system nor the official documentation, but we had the help of an expert researcher instead. His opinion may differ from the opinion of the original designers.

Another issue involved may be the definition of “importance” used by the expert. He was told that the algorithm will try to find the most useful-to-read files, and he was asked to do the same. The difference may reside in that the algorithm assumes a file is worth reading if it is in charge of controlling the flow of the execution, but the expert may think that the files worth reading aren’t from that kind.

As in (Zaidman et al., 2005), the low percentage of false positive may be because those files included in the ranking by the algorithm and not present on the expert’s list may still be important to read to someone different that the expert. A similar explanation can be argued for the false negatives. If we think that we are validating our results against a probably subjective opinion, then the fact that the files not present on the algorithm’s list may not be so important to read to all the programmers is a possibility.

Considering all the above, we can say that the hubiness was affected by the paradigm shift, and thus had a poorer performance on this new environment. The CBO had a similar performance in both environments, and the fan-in, a very simple dynamic measure, had a good performance in this non-OO environment.

Independently of the possible subjectivity issues, the hubiness had a poorer performance compared to the CBO and the fan-in as we can see. This may be due to the particular design of the validations involved in a scanner task. In a typical execution, the system will call several functions where, if a condition is not meet, the function will exit immediately. This alternative to the more intuitive implementation, where the condition must be checked before calling the function to save time and resources, may be adding noise to the measures, especially to hubiness. Taking into account that the hubiness depends both on the fan-in and the fan-out, we can see that some noise may be generated

by this design decision, that is adding incoming calls to several functions that aren't making any real job at all and outgoing calls that are landing on useless functions, making the hubiness of some functions and authority of some others higher than it should.

The CBO had a better performance although it also depends on the fan-in and fan-out. This may be due to that the dependence on these measures is weaker on the CBO than in the hubiness, because the hubiness "transfers" the importance of a function to those related to it, spreading any possible noise, but the CBO just uses the fan-in and fan-out "locally", thus the noise is somehow controlled. Something similar may happen when we look just at the fan-in. Even if some functions are called without being "worthy" enough, the functions that are important will be also called, so the ranking we create will have both important and not-so-important functions.

The distribution of the files found by each of the measures is very interesting. In general terms, the 5 measures placed the files of the expert's list in a very similar order to that ranking, being fan-out and CBO the most accurate measures with 25% of expert's files scrambled each one, followed by hubiness with a 30%. (a file is scrambled when it is placed higher than a file with a greater expert's score). This shows that CBO, fan-out and hubiness are somehow "in touch" with the expert's ranking, i.e. these 3 measures prioritize the expert's files in a very similar way. Since these 3 measures are based on incoming and/or outgoing calls, we can say this heuristic is pretty good if we want to order the relevance of some software artifacts, independently of the paradigm involved.

Summarizing, we can say that the hubiness behaved worse than in the original research with a difference of 40% in false negatives and 20% in false positives, and also had the poorest performance among all the measures used.

4.3.2 SORter

The time the students spent doing the test ranged between 27:54 minutes to 54:58 minutes, with a mean of 39:54 minutes, the lowest time was from a student using SORter and the highest from a student using SeqDev, and the average time of the entire test was 38:06 for SORter users and 41:42 for SeqDev users. This difference of almost 4 minutes represents approximately 1/3 of the time spent in each question, which means that SORter users were 10% faster than SeqDev users to solve the test.

Speaking about the scores, the results varied from 7 points to 10 points, with a mean of 8,58. The average score from SORter users was 8,83 points and 8,33 for SeqDev users. It is clear that this difference is not significant enough, but this happened for reasons that we will explain later on.

Although the difference in time and score may seem too narrow, the presence of SORter had an impact on the performance of the students, helping them to answer the tasks faster and better (in average) than with the help of the other tool. This tells us that our hypothesis may be in the right path, which states that a PC research with good results in an OO context, in this case visualization of execution traces similar to (Scientific Toolworks Inc., 2010) or (University of British Columbia, 2003), can be used as the guidelines to develop a non-OO PC tool that will show also good results in a new environment.

Tool Used	Time				Score			
	Task 1	Task 2	Task 3	Total	Task 1	Task 2	Task 3	Total
SORter	12:40	18:18	15:44	46:42	2	4	1	7
	17:27	06:50	03:37	27:54	2	4	4	10
	14:33	12:52	12:16	39:41	2	4	3.5	9.5
SeqDev	09:05	15:57	11:06	36:08	2	2	3.5	7.5
	10:38	15:42	07:41	34:01	2	2	3.5	7.5
	20:52	16:25	17:41	54:58	3	4	3	10

Table 5 - Times and Scores from Usability Test

This table shows the times and the scores the different students got in the usability test performed for SORter.

In Table 5, each row represents a student, the green numbers represent the best measure (time or score), the yellow the second best and the red the worst measure. Even when SeqDev users had several of the second to last times, the fastest SORter user had a great difference with the rest of the participants, which led to a lower average time. We also can see that two of the lowest times are from the same student, as well as two of the highest times. During the test, the SORter user with the lowest times manifested that the information he needed to solve the tasks was already in the SORter main view, thus he didn't needed any exploration of the parameters involved in the execution. On the other hand, the SeqDev student with the highest times commented that he had problems analyzing all the nodes involved in the execution in order to find what he was looking for.

When the domain expert evaluated the test, he noticed that the first SORter user did not understand completely the tasks he was asked for, which could explain his poor performance both in time and score. Despite this, we can see that the other SORter users had a superior performance, both in time and score, than the SeqDev users.

An interesting outcome can be seen in the second task, which involved comparison between executions: the fastest, second to fastest and slowest time were from SORter

users. Also, we can see that all the SORter users got perfect score and the SeqDev users had a poorer performance, unlike the other two tasks where the scores were more even. The two fastest users opened two instances of the tool, while the slowest one didn't. This shows that being able to have both executions' information at the same time greatly improves the performance when making execution comparisons, action that is impossible to do with SeqDev. The good score performance of SORter users can be due to the easiness of analyzing the entire sequence, which helps you to not miss any detail, thus making the analysis more accurate, unlike SeqDev, which only lets you browse the information by parts, making the analysis of the entire execution more tedious.

Despite the fact that SeqDev users were 1 minute faster in average than SORter users in the first task, all of the participants got a rather low score on this question. This may be because it was the first task they had to answer, so all of them had problems understanding what we were asking them and problems using the tool, regardless they all received an introduction to the use of the tool in a previous session. This reaffirms the fact that the user of any tool must have a complete understanding of how to use it in order to exploit its features and actually improve his performance.

The third question, that was about finding the errors in a sequence, had a good performance for all the students, everyone over 3 points, except for the first SORter user who got just 1 point. This also is explained by the observation of the expert, who stated that this student had problems understanding the tasks. The techniques the other students used to answer this task varied between the tools. SORter users based their answers more on the structural incongruences of the sequence, and the SeqDev users based their answers in the parameters set in the sequence simulator program and not in information presented in SeqDev itself. As a result, both parties had a good performance, but SORter users where 1:30 minutes faster in average than SeqDev users in this question. This shows that having access to the complete structure of the execution at the same time helps when you are trying to find semantic errors in an execution, because it lets you to

analyze the “big picture”, finding wrong patterns in the call tree easier than analyzing the execution by segments.

Although both tools had exactly the same information, SORter users could browse it faster, allowing them to either answer faster or to spend their time preparing a better answer and not searching for information. This help lead to SORter users with better times or higher scores, depending on the personal abilities of the participants.

5. CONCLUSIONS

In this final chapter we will review the results of the experiments, we will analyze their implications and finally we are going to discuss some venues to future work.

5.1 Results

As we detailed in section 4.3.1, all the measures we used to rank files in the HITS algorithm ranked as first priority one of the two most important files according to the expert, but none of them were able to find all the 21 files. CBO missed 9 files, hubiness missed 11, authority missed 10, fan-out missed 9 and fan-in missed 7. There were 6 files that weren't found by any measure, corresponding to the ninth, fourteenth and the last 4 in the expert's ranking, with a priority of 3 to the ninth, 2 to the fourteenth and 1 to the rest according to the 0 to 4 scale used to rank the files.

Reminding the results of SORter reviewed in section 4.3.2, the time the students spent doing the test ranged between 27:54 minutes to 54:58 minutes, with a mean of 39:54 minutes, the lowest time was from a student using SORter and the highest from a student using SeqDev, and the average time of the entire test was 38:06 for SORter users and 41:42 for SeqDev users. This difference of almost 4 minutes represents approximately 1/3 of the time spent in each question, which means that SORter users were 10% faster than SeqDev users to solve the test.

Speaking about the scores, the results varied from 7 points from a SORter user to 10 points from a SORter user and a SeqDev user, with a mean of 8,58. The average score from SORter users was 8,83 points and 8,33 for SeqDev users.

5.2 Implications

After analyzing the results of the experiments we can say that a tool designed to help in the understanding of OO systems is likely to have a poorer performance when it's tested in another environment, thus it can't be directly applied on non-OO systems. This can be explained because the key information used by the tool was highlighted in the original language but not in the new environment's notation (as happened in our case with the classes structure and the hubiness analysis), so the quality of the recommendations made by the tool in the new environment will drop compared with its original study, because the information the tool needed was harder to get.

However, we must admit that this is not a rule, there may be cases when porting a tool will not drop its performance but, as previous research and our findings suggests, in most of the cases when the origin and final language remarks different kinds of information, the port of a tool will not succeed completely.

Despite the above, we can affirm that the research behind PC tools meant for OO systems can be successfully used as a basis to develop PC tools that can get good results in any environment. When a tool doesn't implies any specific characteristic of the target language, and instead it focuses in showing platform-independent information of the system in a way easy to analyze by the programmer, we can evade the notation highlight problem because we are not analyzing the code itself, but the semantic of it. If a researcher wants to port an approach from one paradigm to another, he must focus in adapting the objective of the tool rather than the mechanism to get there, because this mechanism is what can depend on the highlighted characteristics of the language. This conclusion is supported by the results we got with the call stack wrapped with the domain knowledge in SORter, or the results with the patterns architecture proposed in (Cruz, Henriques, & Pereira, 2007).

While it's true that Program Comprehension tools may benefit from the characteristics of a particular language or paradigm, they should not be fully dependent on the chosen environment because, as we just exposed, this can lead to the development of tools that just can show the researcher's theory in the original environment but that aren't scalable, let alone usable as an end-user product.

Another conclusion that can be drawn thanks to the experiments results is that PC tools can't be too generic, because they will help little or nothing to the comprehension task. This is due to the lack of information, both as input to the involved algorithms and as output to the user. If the tool is too generic, we will be missing the opportunity to use any information contained in the language itself or in the problem's domain as input; also we will not know what kind of information is relevant to the code reviewer as an output, besides some generic assumptions made based on metrics from the code or the execution trace.

From the results obtained with HITS we can say that any approach based on metrics doesn't only depends on the target paradigm or language, but also depends on the quality of the software's design. Even when the performance of our port was rather encouraging, the ranking generated was not as accurate as the results of (Zaidman et al., 2005). This was in part due to the change of paradigm, that meant a change in the general structure of the system and the method calls, but also the design of the system had a great impact. The responsibilities in our study case are scattered amongst several files, which enhances unnecessarily the function calls count, adding noise to the metrics used by HITS.

From the usability test ran on SORter we can say that a graphic visualizer will always be an improvement to any understanding task in general, because it allows the reviewer to browse large amounts of information faster and easier than, for example, plain text like documentation or source code.

Another important conclusion is that a previous training is necessary to use any PC tool, because we will only see an improvement in the performance of the programmers, let it be in time or accuracy, when they are familiarized and feel comfortable with the tool at hand.

5.3 Future Work

The field study done in the context of this work shows that there is a necessity of moving the research efforts towards other paradigms besides OO. Following this idea, a new venue to research could be a study to determine in a first stage the most popular programming language in the industry and/or the most common language already present in the industry, for example languages of legacy software. With this information, the next step is to either discuss the portability of common PC approaches to these languages or to define frameworks that allow the design of tools regardless the language involved. A good example of the latter is the patterns framework proposed for the tool Alma (Cruz et al., 2007), which defines seven statements or "patterns" that any programming language should have and designs a tool based only on that, which makes it independent of the target language.

Regarding the tools developed in the context of this work, they can be improved both in interface and model representation, given that they were designed as concept proofs and not complete products. The different features of HITS + fe-Tree can be integrated in a better way, since as it is now it still looks like a mash-up of tools and not a single tool. On the other hand, an API can be developed for SORter in order to allow programmers to easily use this tool in their problem's domain.

Regarding the experimental design, our hypothesis can be target of further validation using different study cases, in which more documentation, support and programmers are available. With these conditions, the experiment can be conducted with larger groups, so the results could be statistically relevant.

REFERENCES

12th Course of the International Zurich Magnetic Resonance Education Center. (2001, February 12). Gyroscan-NT Pulse Programming.

Aho, Antoniol, & Guéhéneuc. (2008). CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. Presented at the The 16th IEEE International Conference on Program Comprehension.

Antoniol, G., & Guéhéneuc, Y. G. (2005). Feature identification: a novel approach and a case study. *2005. ICSM'05. Proceedings of the 21st IEEE International Conference on Software Maintenance* (págs. 357–366).

Béron, M., Henriques, P., Pereira, M. J., & Uzal, R. (2007). Static and dynamic strategies to understand C programs by code annotation.

Bohnet, J., Voigt, S., & Doellner, J. (2008). Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-Focused Views on Execution Traces. *2008 The 16th IEEE International Conference on Program Comprehension* (page 268-271). Presented at the 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, The Netherlands. doi:10.1109/ICPC.2008.21

Booch G. (1986). Object Oriented Development. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*.

Brooke R. (1975). A model of human cognitive behavior in writing code for computer programs. *Proceedings of the 4th international joint conference on Artificial intelligence* (Vol. 1, págs. 878-884). Presented at the International Joint Conference On Artificial Intelligence, Morgan Kaufmann Publishers Inc.

Campos, Cortazar, Eterovic, Tejos, & Irarrazaval. (2009). Visualization Tools for Understanding a Complex Code from a Real Application.

Carey, M. M., & Gannod, G. C. (2007). Recovering Concepts from Source Code with Automated Concept Identification. *15th IEEE International Conference on Program Comprehension, 2007. ICPC'07* (págs. 27–36).

Chidamber, S. R., & Kemerer, C. F. (2002). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, *20*(6), 476–493.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering (TSE).*

Cornelissen, B., Zaidman, A., Van Rompaey, B., & van Deursen, A. (2009). Trace visualization for program comprehension: A controlled experiment. *Proceedings of the 17th International Conference on Program Comprehension* (págs. 100–109).

Cruz, D., Henriques, P., & Pereira, M. J. (2007). Constructing program animations using a pattern-based approach.

GrammaTech. (2007). CodeSurfer. August 11, 2010, http://www.grammatech.com/products/codesurfer/overview.html

Green T.G.R., G. D. J. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*(1), 31-48.

Intland Software. (1999). CodeBeamer. August 11, 2010, http://www.intland.com/products/cb/overview.html

Lehrstuhl für Softwaretechnik - Universität des Saarlandes. (2010). Delta Debugging. *Lehrstuhl für Softwaretechnik - Universität des Saarlandes*. August 12, 2010, http://www.st.cs.uni-saarland.de/dd/

Meyer. (2004). *Object Oriented Software Construction* (2o ed.). Prentice Hall.

Oliveira N. (2010). *Improving Program Comprehension Tools for Domain Specific Languages*. Braga, Portugal: University of Minho, Informatics Department.

Price, B. A., Baecker, R., & Small, I. S. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, *4*(3), 211–266.

infotectonica sa. (2006). Juliet: Instant Code Comprehension for Java Programmers. *infotectonica - Intelligence Augmentation For Java Programmers*. Recuperado Agosto 13, 2010, a partir de http://infotectonica.com/juliet/

Scientific Toolworks Inc. (2010). Understand - Source Code Analysis & Metrics. *SciTools - Maintain your Software*. Recuperado Agosto 11, 2010, a partir de http://www.scitools.com/index.php

Shilling, J. J., Stasko, J. T., Graphics, V., & Center, U. (1992). Using animation to design, document and trace object-oriented systems. *Georgia Institue of Technology, Technical Report GIT-GVU-92*, *12*.

Storey, Wong, & Müller. (2000). How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, *36*.

The Eclipse Foundation. (2001). Eclipse. January 20, 2011, http://www.eclipse.org/

The Unravel Project. (1999). The Unravel Program Slicing Tool. *The Unravel Project*. August 12, 2010, http://www.itl.nist.gov/div897/sqg/unravel/unravel.html

University of British Columbia, O., Inc. (2003). AVID - Component-based Visualization of the Execution of a Java System. *Canadian Consortium for Software Engineering Research (CSER)*. August 13, 2010, http://people.cs.ubc.ca/~murphy/AVID/

VisiComp Inc. (2004). RetroVue - The Total Recall Debugger. *VisiComp, Inc.* August 12, 2010, http://www.visicomp.com/index.html

Weiser M. (1981). Program Slicing. *Proceedings of the 5th international conference on Software engineering* (págs. 439-449). Presented at the International Conference on Software Engineering, San Diego, California, USA: IEEE Press.

Wiedenbeck S. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *Academic Press*.

Zaidman, A., Calders, T., Demeyer, S., & Paredaens, J. (2005). Applying webmining techniques to execution traces to support the program comprehension process. *Ninth European Conference on Software Maintenance and Reengineering, 2005. CSMR 2005.* (págs. 134–142).

Zeller, A. (2000). From Automated Testing to Automated Debugging. *Uni Passau, Feb*.

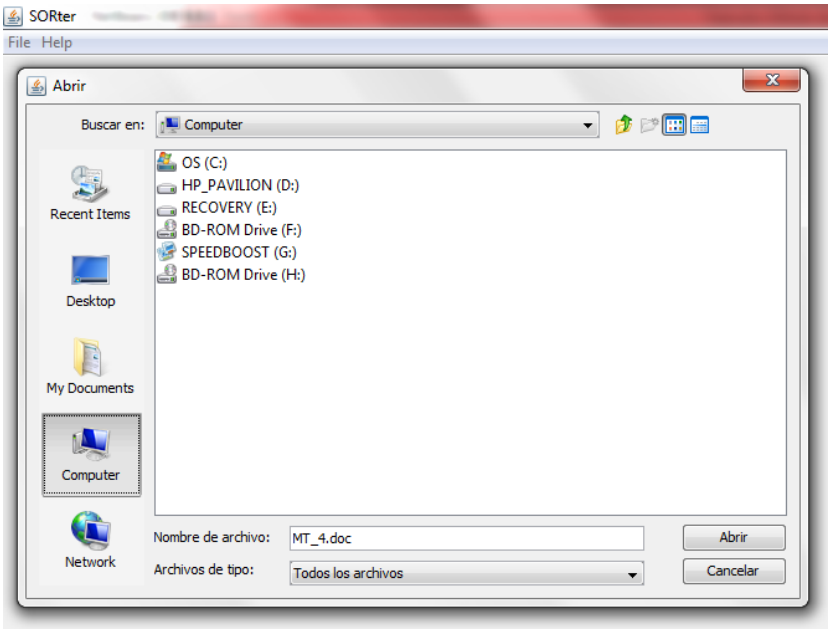
APPENDIX

A) SORter Manual

SORter is a visualization tool designed to understand systems that implements sequence of actions. The input of the tool is a text file that represents the different actions the system does to achieve its goal. The input file can either be written by the user himself or you can use I/O instructions in the key sections of the code to create the text file.

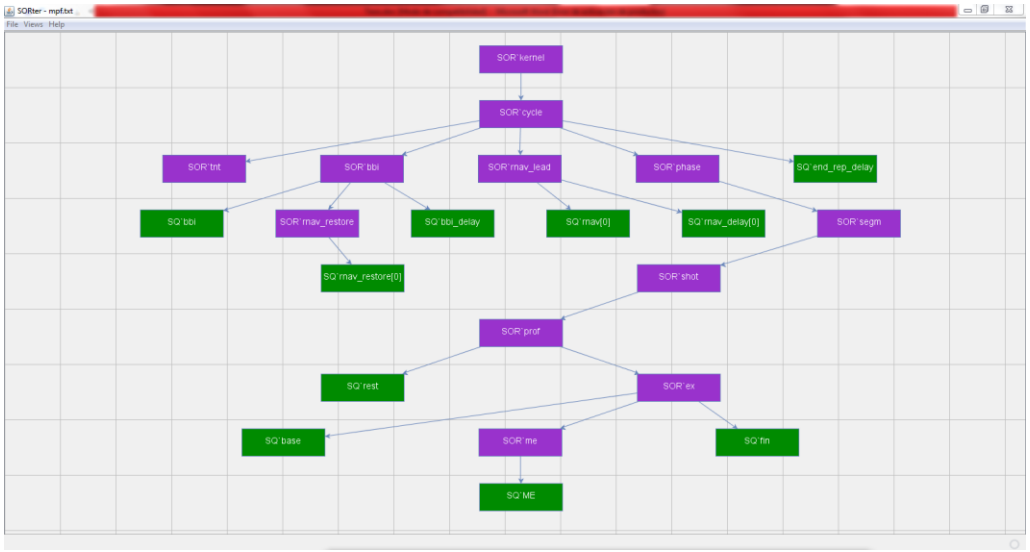
```
SORterInput=============================================================
SOR`kernel:ref_obj = [ SOR`cycle (0)     ], dur = EMPTY
SOR`kernel:dur     = 1542.8572
SOR`kernel:ref     = 578.6880
SOR`kernel:dur2    = 964.1692
SOR`kernel:min_dur = 1542.8572
SOR`kernel:sar     = 11420.5918
SOR`kernel:rf_tot_gating_dur  = 114.8064
SOR`kernel:rf_max_b1= 25.7249
#1 *SOR`cycle       (rf enabled)
#1  SOR`cycle       (rf enabled)
    SOR`cycle:ref_obj = [ SOR`phase (0)     ], dur = EMPTY
    SOR`cycle:dur     = 771.4286
    SOR`cycle:ref     = 578.6880
    SOR`cycle:dur2    = 192.7406
    SOR`cycle:min_dur = 771.4286
    SOR`cycle:sar     = 5710.2959
    SOR`cycle:rf_tot_gating_dur  = 57.4032
    SOR`cycle:rf_max_b1= 25.7249
    #1  SOR`tnt         (rf enabled)
        SOR`tnt:ref_obj = [ undefined (0)     ], dur = EMPTY
        SOR`tnt:dur     = 0.0000
        SOR`tnt:ref     = 0.0000
        SOR`tnt:dur2    = 0.0000
        SOR`tnt:min_dur = 0.0000
        SOR`tnt:sar     = 0.0000
        SOR`tnt:rf_tot_gating_dur  = 0.0000
        SOR`tnt:rf_max_b1= 0.0000
    #1  SOR`bbi         (rf enabled)
        SOR`bbi:ref_obj = [ undefined (0)     ], dur = EMPTY
        SOR`bbi:dur     = 525.3918
        SOR`bbi:ref     = 0.0000
        SOR`bbi:dur2    = 525.3918
        SOR`bbi:min_dur = 525.3918
        SOR`bbi:sar     = 1039.5174
        SOR`bbi:rf_tot_gating_dur  = 23.8728
        SOR`bbi:rf_max_b1= 15.7127
```

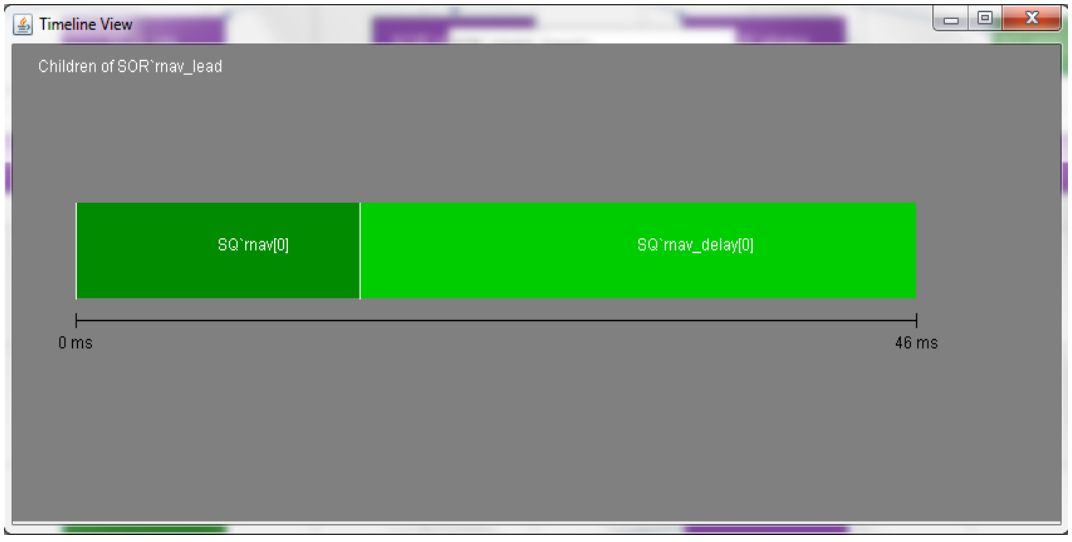
To analyze an execution open the input file through the tool's menu (File→Open) or using the hotkey Ctrl+O. A new window will pop up where you can select the input file.



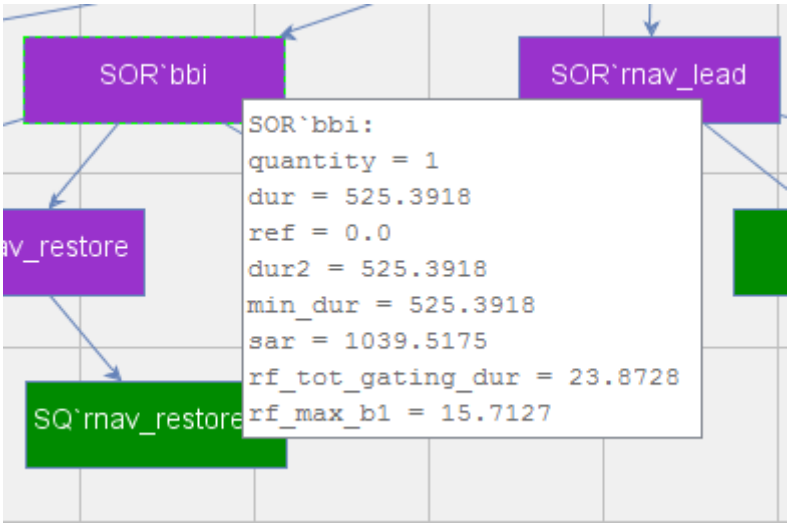
Once the file is open two views will be

displayed, a hierarchical view that will allow the user to see the general outline of the execution and a timeline that will help him to understand in detail each action.

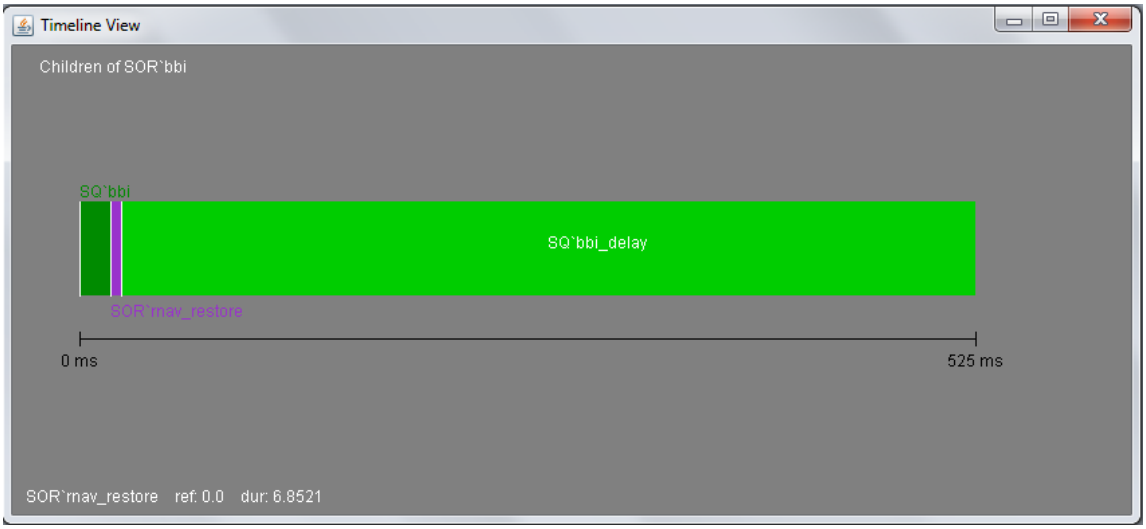




In the hierarchical view the user can right-click on any node to see a pop-up with detailed information of that action (duration, start and end time, etc).



That action will also update the timeline, showing in scale the duration and start/end time of all the children of that node, i.e. all the actions that derive of the first one. If you left-click on the name of an action in the timeline the values of it start time and duration will appear in the lower left corner.



The timeline also can be updated by the upper menu (Views option), when the user can see two different groups of actions:

- All the SQ labeled sequences (custom for the GyroScan System)
- All the actions of a determined deepness level in the hierarchy tree.