STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXII**, Number 2, 2017
DOI: 10.24193/subbi.2017.2.06

IMPROVING PROGRAM COMPREHENSION THROUGH DYNAMIC CODE ANALYSIS

ROBERT FRANCISC VIDA

Abstract. Most of the software that is currently being developed in the industry tends to be very complex and it is safe to assume that a lot of future software projects will keep escalating in their complexity. This means that developers need to keep track of numerous aspects of their system at all time and that new additions to the team will have difficulty adjusting to projects. In this paper, we will propose a set of techniques that combine already existing dynamic code analysis concepts to resolve the aforementioned problems, which can be summarized as being program comprehension difficulties. To do this we introduce a few novel software analysis and visualization techniques that facilitate program comprehension. The approach proposed within this paper allows for easy identification of semantic information, data available at execution time, which might be difficult or even impossible to portray in an easy to understand representation using existing software visualization techniques. During this paper we will be considering traditional object oriented programming languages, however these ideas should be useful in the context of other programming paradigms as well.

1. Introduction

It is safe to say that currently, all over the world, there are many software systems being developed for different purposes. Obviously, the size and scope of these projects may vary but all of them have a certain degree of complexity, which, as stated in [10], is one of the leading causes of failure in the industry. That being said, we believe that if the developers have a good understanding of the system at all times, they will be able to minimize complexity growth

Received by the editors: November 30, 2017.

2010 *Mathematics Subject Classification.* 68N01, 68M20.

1998 *CR Categories and Descriptors.* K.6.3 [**Management of computing and information systems**]: Software Management – *Software maintenance*; D.2.5 [**Software Engineering**]: Testing and Debugging – *Tracing*.

Key words and phrases. Dynamic code analysis, program comprehension, software visualization.

as new features are added or maintenance is performed, this in turn reducing the risk of project failure.

The goal of this paper is to explore new techniques that might reduce the change of developers needlessly adding complexity to their software system. This is done by aiding them to better understand the behavior of their system while also helping new additions to the team gain knowledge about the system in an easier manner. This is called program comprehension. Most of the techniques that we will present rely on dynamic program analysis and have been designed specifically for object oriented programming, however they might be relevant for other programming paradigms as well.

The remainder of this paper is organized as follows. Explaining what program comprehension is and why we need it in Section 2. Section 3 presents dynamic program analysis and why it was chosen over its static counterpart, as well as describing the importance of software visualization and how it can be used. In Section 4 we review existing solutions for software visualization and list a few tools which can be used in order to enhance program comprehension and discuss their strengths and shortcomings. We present our original approach and provide a few detailed examples to better explain the techniques along with their concepts in Section 5. In Section 6 we mention a prototype that facilitates some of the techniques discussed and finally, in Section 7, we have the conclusion of this paper along with a few ideas for potential future work.

2. Program Comprehension

One of the most important things to consider before altering code is how much of it is actually understood by the person performing the change. It is imperative for the developer to fully understand, in as much detail as possible, how the program works before attempting to modify it since the chances of damaging the existing system are direct proportional with the size of the system, usually very high. This is also explained in [2]. The damage can range from the obvious bugs and inconsistencies to the more annoying and hard to identify ones such as performance, security and reliability.

In [11], the author presents the usefulness of program comprehension by describing what activities are required during maintenance and evolution tasks. The common activity across these tasks implies understanding the system. This should speak volumes about the importance program comprehension at all stages of the development cycle.

One of the most obvious methods of keeping a program easy to understand is to keep the code clean by the means of code review and constant refactoring sessions. The problem is that even if the code is clean, sometimes it can be

very difficult to see the bigger picture, like how some components interact with each other on a bigger scale or in stretched out scopes. In this case, the obvious answer is to keep the system well documented. This is relatively easy to achieve since all you waste is a bit of time and can work wonders in a lot of cases. That being said, having too many system descriptors, be it explicit (documentation) or implicit (clean code) and still expect high program comprehension from the developers is a bit too much to ask, especially if they are relatively new to the project. It is very difficult for people to understand a system just by reading about it.

The solutions proposed to solve this problem, or at least alleviate its effects, are based on the fact that people are more susceptible to understand something if they are actively involved in the process of researching [9]. This is in contrast to more passive methods like looking through code or reading documentation, where the person researching can only think what is happening at any given moment. That being the case, we will look over different techniques through which, by using dynamic program analysis, one can gain a better insight over what happens within the program.

3. Dynamic program Analysis

Dynamic program analysis is a technique through which one can analyze the different properties of a program while it is executing. At times, static analysis has been used to analyze the dynamic behavior of programs since it was easier and did not require the execution of the program either. This, however, does not yield results as precise as actually doing dynamic analysis would. Also, as programming languages have evolved to running in more dynamic environments, the presence of features like dynamic binding, polymorphism and threads have made static analysis quite ineffective. This is because static analysis can only check what is present. The packages that may be loaded dynamically when running the application might not be present until the execution, so this shortcoming is understandable.

The two techniques mentioned earlier, dynamic program analysis and static program analysis are complementary, each having their own strengths and weaknesses. The static analysis examines the source code rigorously at compile-time. Relevant techniques include data-flow analysis, which analyzes how variables change their values through the execution flow, symbolic execution, which determines what input values case each part of an application to execute and dependence analysis. Dynamic analysis looks at an application while it is running and analyzes data obtained from that. Two commonly used techniques are assertions, which are simple checks inserted within the source code itself in order to check various things and coverage analysis which analyzes

the flow of the execution while an application is running [4]. In this paper we will look at dynamic program analysis and explore to some extent the two methods mentioned earlier.

Dynamic analysis can help guide the development process towards producing a solution that behaves, within the realm of possibility, as intended, as well as aiding the developers in enhancing and optimizing an already working system. By analyzing the relationships between independent threads or duration of method calls as well as the context they are called from we can easily devise solutions that might improve overall performance. This is also clearly stated by Thomas Ball in his article [3].

3.1. **Software Visualization at Runtime.** Software visualization refers to displaying information about or related to the software system in a visual-oriented manner so that it is easier to understand and interpret [6]. The information type that can be used can range from the architecture of the system, how the code is structured, to its runtime behavior, algorithm behavior.

In order to extract runtime information without altering the source code, one could use profiling, which is a type of dynamic program analysis. Through it we can obtain information like space or time complexity, method calls and other statistics that are generated while running the application. Using this information it is fairly easy to construct an execution graph or calculate the frequency and duration of subroutines. It is on this type of analysis that the techniques presented in this paper are based on.

After the desired information has been obtained, the next important step is deciding on how the information shall be displayed. One obvious solution would be to display the flow of execution in the form of an UML Sequence Diagram. This would be quite suitable since this type of diagram is very intuitive and exposes the information and component interactions nicely, however there are a two fatal drawbacks. The first one is that if the execution flow complexity is too great, it will become very difficult for a user to follow through, and the second is that this type of diagram was not created with the idea of multiple threads of execution.

In order to identify the most suitable form of visualizing software runtime data we need to take a look at what developers usually use to gain knowledge regarding the system. That being said, the built-in debugger that most IDEs (Integrated Development Environment) have would fit the description perfectly. Representing the execution flow in a tree-like manner, which if going from a leaf to the root will look like the common execution stack, would be ideal since the user will already be familiar with the format.

4. Related Work

Trying to gain program comprehension by analyzing the execution flow is not a new idea, however the approach that we took in this paper is. In [12] we see an approach to compare different execution traces to identify changes within execution code, order and duration. The difference between this paper and ours, is that we try to focus on providing as much information as possible regarding a execution trace and present it in a easy to understand manner. Comparing two execution flows is also presented here, however it is not the main issue we want to tackle.

In [8] we see a similar approach to ours, applying different program analysis techniques on the software in order gain some understanding of it. Technically speaking, it takes it a step further by using both static and dynamic code analysis, whereas we only use the latter. The techniques that we propose in this paper can be used not only to understand code someone else wrote but to gain insight into ones own code as well. The other slight difference is the manner in which data is presented, we chose to present the data in a tree-like manner while in the aforementioned article they seem to have chosen to go with graphs.

Along the years there have been numerous tools that have aided developers in analyzing the software they develop, assuring them that they are on the right track and reducing the possibility to introduce faulty or algorithmically incompatible features into their system.

One might argue that a debugger can be considered a tool for facilitating program comprehension by observing the execution flow, however it has a very big flaw. Since their aim is to inspect the code by stopping the execution at certain points called breakpoints, they often bring the application in a state in which it couldn't naturally be in. This is an especially serious matter in situations where there are multiple threads or scheduling components.

VisualVM is a visual tool written in and for Java that uses lightweight profiling to extract statistics regarding an application during its runtime [13]. The application is very easy to use and can connect to running applications at any time. The main drawback of this tool is that it only uses a flat profiler. This means that it only gathers data regarding memory consumption, execution duration and execution time. This is good if someone needs a quick overview of the system during its execution or if they are looking for memory leaks, however it lacks context. That being said, it is impossible for the developer to reason the behavior of the system against these statistics.

Gprof specializes on call graph executions. Call graphs can be both static, which takes into account all possible routes and does not require the application to be running, and dynamic, which takes into account only executed

methods. This means that it is able to assess the cost of routines accurately [7]. Because of this it is easy for the people running the analysis to see the methods that were called during the execution and also their ordering, giving us a context of them. Seeing the path the execution took, can give developers a few hints where something went wrong or where optimizations might be possible.

Aprof is a Valgrind tool designed to help developers identify inefficiencies in code [5]. It is input-sensitive, that means that on top of call graph, it takes into account the input for methods and measures their performance based on the workload received. This is very important because the analysis allows the developers to pinpoint the exact location where the execution ran off track.

5. Proposed Concepts and Techniques

The techniques presented within this paper are very straightforward and previous knowledge regarding dynamic program analysis is not required in order fully understand the ideas behind them. Before we discuss the techniques themselves we should first define a few concepts which will be used. Some of these concepts are not new by any means, while the others can be seen as being built on top of existing ideas. Either way, it is important to understand them since they are the foundation for the techniques that we will discuss later on.

Since we will be looking at techniques through which to extract and present the execution flow of an application we will need to analyze the simplest component that we can relate to. That being said, considering that we are targeting object oriented programming, this would be the class method.

5.1. **The Method Structure.**

An application is usually composed of multiple classes. Each of these classes have methods of the format $Method = (L_{input}, L_{instructions}, V_{output})$, where: L_{input} is the list of input arguments which can be empty, $L_{instructions}$ is the list of instructions within which can also be empty and V_{output} is the optional return value from the method. It is important to keep in mind that the instructions may contain calls to other methods. The best format to express the method execution, considering our needs, is a version that offers the following components: input arguments L_{input} with call time t_0, output value V_{output} with return time t_f and a set of the method instructions that contains only other method calls $L_{method} = (i | i \in L_{instructions}$ such that i is a method called during execution). The format for the method execution will look like this:

$$Method_{execution} = (t_0, L_{input}, L_{method}, t_f, V_{output})$$

5.2. **Concepts.** Each of these concepts are independent to each other, this is important since it means that they can be used separately or in combination to each other. This allows us to gather more specialized information regarding our system, information that is more relevant to our goal. The concepts that we considered in our approach are:

> Call stack or Call tree - sequence of calls presented in a stack or tree layout. The main reason for choosing the approach of using a stack layout is because of the familiarity developers have with stack traces used when debugging.
> Selective focus - in order to minimize the impact on the running application it would be optimal to focus attention on only parts of it.
> Context information - sometimes, having information on what each method call starts with and produces is for the process of understanding what exactly is happening to the application while the executing.
> Selective focus using context information - this is selective focus enhanced with the knowledge obtained from analyzing the context. Basically only recording methods when certain conditions are met.

5.2.1. *Call Stack/Call Tree.* The call stack or call tree is purely a visual concept through which one can depict the execution flow of an application. Having a clear and intuitive way of checking the execution steps of an application is a crucial aspect. It is mainly though this that the users observe how the program unfolded, thus it is crucial to the process of understanding the system.

The decision to organize and present the method calls in a tree-like manner was made with the purpose of having the developers already be familiar with the representation since it works similar to how a stack trace works in debug mode, just a bit more hierarchical.

As the root node we will have the signature of the method being called. This will include the method name and the types of parameters it accepts. The first child will be composed out of the parameters sent when calling the method. If no arguments were sent, then this node should not be present at all since it would be irrelevant. The children of the node will be the elements of L_{input} component mentioned earlier. The last node will represent the return statement of the method, this node should always be shown because this means that the method finished successfully. If the method returns a value, the value will also be shown. This would be V_{output}. In between the first and last node will be the nodes of all methods called from within the current method, sorted chronologically. These nodes will be method nodes themselves and there should be one for each element of L_{method}.

```
void childFunction(){...}
void parentFunction(){...}

int main() {
  if(fork() == 0) {
    childFunction();
  } else {
    parentFunction();
    wait();
  }
  return 0;
}
```

FIGURE 1. Call stack code

	Thread 1		Thread 2
0	main()		
10	Parameters: ()		
11	fork()		
15		15	childFunction()
18	parentFunction()	18	
20		20	Parameters: ()
21	Parameters: ()	21	
23		23	Return
25		25	Retrun 0
26	Return		
27	wait()		
30	Return 0		

FIGURE 2. Call timeline visual representation

All of the nodes will have a time associated with them, this will be represented by a number in a column on the right hand side. This will be very useful since it offers information on the duration of the calls.

In case there are multiple threads running, there should be separate trees for each thread. The nodes of the threads should be intertwined, with empty nodes representing that something happened on another thread at that time.

In Figure 1 we can see a sample code of a program that has two threads and each of them goes on to call a different function. Figure 2 depicts an execution of this code. We can see where each function call starts for each thread. It is clear where the function execution overlapped and where they stopped.

A stacktrace displays the order in which methods were executed and a common profiler can provide insight into the context of the execution environment. With this format we gain both at the same time. We can see contextual information integrated within the execution order of the methods.

5.2.2. *Selective Focus.* Instead of capturing the execution of every function call on every thread, it may sometimes be desired to only focus on a certain thread, or a certain group of functions. Applying such filters on the profiler will drastically improve performance and reduce the impact that the profiler has on the analyzed application.

The selective focus concept provides a good solution for reducing unnecessary analysis on portions of code that are of no interest to the developer. This also allows for other analysis concepts that consume more processing power to be used without the fear that they might disturb the natural flow of execution

```
void visibleChildFunction(){...}
void visibleParentFunction(){...}
void ignoredParentFunction(){...}

int main() {
  if(fork() == 0) {
    visibleChildFunction();
  } else {
    ignoredParentFunction();
    visibleParentFunction();
    wait();
  }
  return 0;
}
```

FIGURE 3. Selective call timeline code

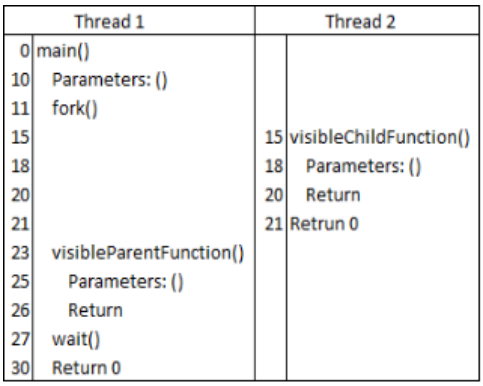

FIGURE 4. Selective call timeline visual representation

of the software too much. It is undeniable that if the developer adds a lot of resource consuming analysis concepts and if the application is multithreaded, then there is a good possibility that the execution will go into a unique state which would not be possible under normal circumstances.

This is not a new concept, there are a lot of tools that have ways to selectively choose what parts of the application to analyze, however this method differs from the way you define these parts and the manner in which the report is generated at the end of the analysis.

In Figure 3 we have a sample code of a program that has two threads and each of these threads will call their own functions. Figure 4 depicts a potential execution of the code previously stated. In this certain representation, the person running the analysis decided to ignore the method ignoredParentFunction() and so it is not represented within the timeline.

To take full advantage of the capabilities of this analysis method, we strongly suggest implementing the filtering system in a dynamic fashion, by this we mean being able to specify target methods through a mechanism similar to regular expressions. This is not hard to do and one would gain the ability to mark the functions that are to be analyzed at runtime. This is imperative if the developer wants to intercept function calls even if they were declared through reflection.

5.2.3. *Context Information.* This concept is concerned with recording the data that is used in inter-method communication. This covers both input and output data, however the amount of how much data to record should be kept in mind. What this means is that if for example we have a class as input data,

we need to specify how deep within its fields we will record. If the class has another class as field, and so on, there should be a stopping point to reduce the stress of the analysis.

All of the data can be stored in a context along with their reference id so we can observe the changes made to an entity through the entire execution. This would allow the users to easily follow data modification during the application execution. It is important to note, that this concept is similar to dynamic program slicing [1], which is a technique that gathers all the statements that changed the value of a variable during an execution. On the other hand, our technique is able to identify the changes done to the object itself. So they are similar in aim, which is to study the evolution of the application state, but have different approaches on how to do this.

The context information concept is one of the most resource-consuming of all of the ones proposed within this paper, however it also is the one that gives the most detailed insight about what happened within the code because it can clearly display all input and output values for each function.

5.2.4. *Conditional Focus.* This concept builds on top of the previous mentioned concept, selective focus, by adding awareness regarding the context. This means that after we filter the parts of the program we want to focus on, we can go even further and add that only when specific input values are passed should we inspect the section. The overhead added to the execution might not be very appealing, however it is well worth the sacrifice in order to have a way to add this sort of flexibility.

5.3. **Techniques.** Next, we will use the concepts defined earlier and combine them so that they will aid us in our goal of understanding the program better. All of these techniques assume that we use a call stack as a way of presenting the information.

5.3.1. *Basic Analysis Technique.* The first combination will be very straightforward and somewhat predictable, we will use selective focus and context information. By using selective focus, we reduce the strain put on the application by the analyzer and by using context information we expose detailed information about the piece of code that we are interested in. This combination is important since with this the developer can gain in-depth knowledge over the part of the application that he desires. Illustrating a call stack/tree of the methods that were executed and the data received, changed and produced. This combination can also be used as an advanced form of logging.

5.3.2. *History Technique.* This technique is actually built on top of the previous one, however the step is in a horizontal direction. What this means is that

it does not go deeper down into extracting more data or filtering the inspected scope of the program, but simply keeps track of multiple executions and attempts to compare them. This is important since this way the developers can examine the evolution of the behavior of a program. They can also execute the same steps over and over again, in order to check the consistency and reliability of multithreaded sections. The visual representation is not difficult to understand, this makes it is easy to present to non-technical users and explain how it all works. It eases the communication bridge between two groups of people that usually have difficulty explaining their point of view to the other.

5.3.3. *Checking Technique.* This technique would more likely be used for testing purposes rather than program comprehension. It is very possible to use a slightly altered version of the conditional focus concept to check at all stages that different values do not pass through certain areas of the code. Using such a method on a system would seem as though it was attempted to add formal verification on top of an already existing system. This is a strange approach since formal methods are performed before any code is written, however this method is worth mentioning since there are some situations where such approaches might be needed.

5.4. **Threats to validity.** It is important to keep in mind that the concepts and techniques presented have not been proven to be a definite improvement over other similar tools nor do we claim them to be. The techniques were devised in order to explore new ideas in the domain of program comprehension and are still in an experimental state at this moment.

An aspect that is quite concerning to the validity of these techniques is scalability. These concepts were only tested in environments of small sized applications that did not make use of too many execution threads of the same. This concern relates to both execution and visualization issues. By this we mean that the tool might behave faulty when a larger application is analyzed, but also that the visualization mechanism might prove to be less suited when too many points of interest need to be shown at the same time.

Another thing to keep in mind is that different programming languages will have different instrumentation limitations. In a few cases these techniques might actually be impossible to implement.

6. Working Prototype

Most of the concepts and techniques presented within this paper have already been implemented into a stable prototype. Written in Java, by the use of instrumentation it is able to observe other Java applications while they are being executed without affecting the normal execution flow too much. The

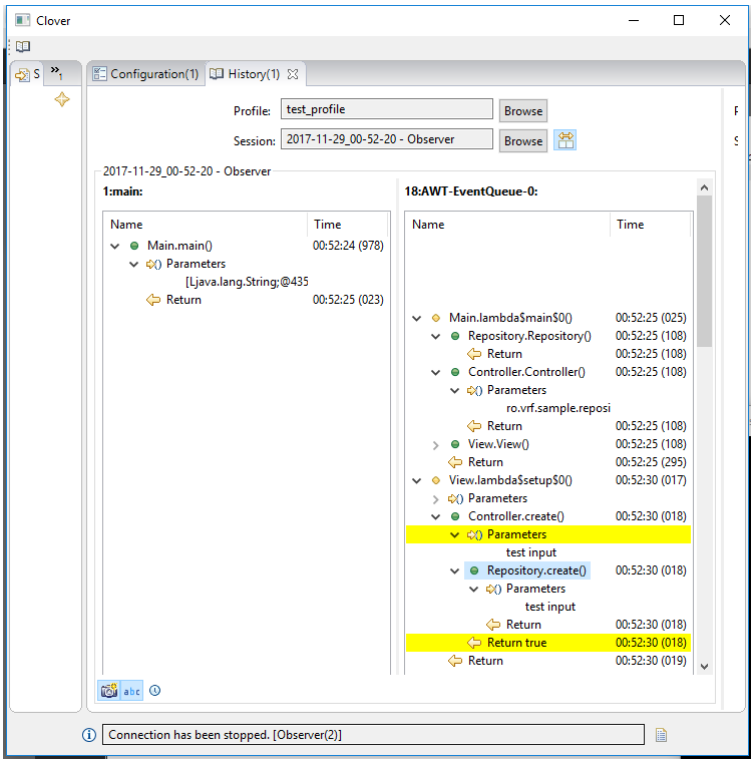

FIGURE 5. Screenshot of the prototype

only impact on the inspected application is that the overhead added by the analysis itself, by this we mean the mechanism through which we extract the data, so the execution threads might slow down a bit. The prototype is able to handle multiple execution threads and structures the flow into a hierarchical manner.

A few of the concepts described in this paper were only partially implemented or do not have the flexibility previously described, the reason behind this is that it is only a prototype meant to show the appeal of such a tool. A noteworthy but not necessarily critical flaw for the tool is the fact that it is unable to inspect the core classes because they are being used in order to extract the data from the analyzed application. The reason we say it is not critical is because one would normally use this application to analyze their own code. In order to gather as much data as possible, we recommend that the instrumentation process starts as soon as possible, exactly when the target application is started would be ideal. The reason behind this is that although

the tool is able to analyze already running applications, it is limited to classes that have not been loaded, by this we mean those that have not yet been used.

In Figure 5 we can see the execution flow of an application that has two threads. The two tree structures depict the methods called from each thread as they are called, each having the identifying name of the thread above them. The root nodes represent the first methods called that respect the filtering conditions set before the analysis began. Whenever there are parameters sent to the methods, a child node containing a list of parameters will be present. Next, if there are other methods called from this method, they will be indicated through separate suggestive nodes. The last child node of a method node will be the return statement that will also indicate the return value if there is any. All nodes, except for parameter nodes and their children, display the time at which they occurred, this way one could easily tell how long the method took to execute. We believe that this way it is easy to see crucial information regarding methods, such as access control modifiers, input parameters, entry and exit time points as well as method calls performed within. However, when there are many chained methods it might be difficult to keep track of the exact location within execution tree. In order to aid the user in orienting themselves within the execution tree, we highlighted with yellow background the entry and exit point of the currently selected method.

7. Conclusion and Further work

From all the information presented in this paper, it is easy to understand the importance of program comprehension and why it is imperative for it to be as high as possible. The method through which this is done is not particularly important, however by using dynamic program analysis you get to observe the application in its most crucial state, at runtime. By directly observing how the application behaves during the execution you get to see how it reacts, no need for speculation, we can see exactly how all parts come together and work with each other.

We have presented various techniques through which one might enhance the experience of gaining or maintaining program comprehension regarding an application along with a working prototype that makes use of these methods. It is important to note that all of these techniques require the user be engaged in the analysis task so that the process of understanding the application can progress more naturally.

In the future we plan to further refine the concepts and techniques previously mentioned as well as extend them to provide more customizable and relevant information to the developers or the interested users. Pursuing other techniques is not of the table. There were a few other ideas that did not make

it into this paper for various reasons. For example a technique through which one would be able to identify hidden dependencies, detect places where design patterns should be implemented or determine if two classes belonging to different components are connected to each other to tightly when they shouldn't (high coupling).

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[2] Usman Akhlaq and Muhammad Usman Yousaf. Impact of software comprehension in software maintenance and evolution. Master's thesis, Blekinge Institute of Technology, 2010. Chapter 8.

[3] Thomas Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[4] Mario Barrenechea. Program analysis. `https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/barrenecheamario.pdf`, . [Online; accessed 5-September-2017].

[5] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *SIGPLAN Not.*, 47(6):89–98, June 2012.

[6] Denis Gracanin, Kresimir Matkovic, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering, A NASA Journal*, 1(2):221–230, September 2005.

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[8] Wilhelm Kirchmayr, Michael Moser, Ludwig Nocke, Josef Pichler, and Rudolf Tober. Integration of static and dynamic code analysis for understanding legacy source code. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 543–552, 2016.

[9] Michael Prince. Does active learning work? a review of the research. *Journal of Engineering Education*, 93(3):223–231, 2004.

[10] Roger Sessions. The it complexity crisis: Danger and opportunity. Technical report, ObjectWatch, 2009.

[11] Priyadarshi Tripathy and Kshirasagar Naik. *A Practitioner's Approach, Software Evolution and Maintenance*, chapter 8. John Wiley & Sons, Inc., New York, NY, USA, 2014.

[12] Jonas Trmper, Jrgen Dllner, and Alexandru C. Telea. Multiscale visual comparison of execution traces. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 53–62, 2013.

[13] Visual vm. https://visualvm.github.io/ [Online; accessed 12-December-2017].

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: `robertv@cs.ubbcluj.ro`