Automatic Verification of Time Behavior of Programs

Giovanni Liva
Supervisor: Univ. Prof. Dr. Martin Pinzger
Institue of Informatics Systems
University of Klagenfurt, Austria
giovanni.liva@aau.at

ABSTRACT

Automatic verification of software could save companies from huge losses due to errors in their programs. Existing techniques to prevent and detect errors are mainly based on imprecise heuristics which can report false positives. Formal verification techniques are an alternative to the heuristic approaches. They are more precise and can report errors with higher rigor. However, they cannot be directly applied because current programming languages have no defined semantics that specifies how the source code is interpreted in the execution of programs. Moreover, no existing work tries to develop a semantics for the *time* domain. The target of this thesis is to provide a verification framework based on a defined time semantics that can help developers to automatically detect time related errors in the behavior of programs. We define a time semantics that allows us to ascribe a meaning to source code statements that alter and use time. Based on the time semantics, we develop an approach to (i) automatically assert time properties and (ii) reverse engineer timed automata, a formal model of the time behavior that is amenable for verification. We plan to evaluate our approaches with quantitative and qualitative studies. The quantitative studies will measure the performance of our approaches in open source projects and the qualitative studies will collect the developers' feedback about the applicability and usefulness of our proposed techniques.

1 PROBLEM STATEMENT

Automatic verification of software could save companies from huge losses [5]. There are many examples where a single bug has cost companies millions of dollars although, many of these bugs could have been prevented. Several approaches have been presented [2, 4, 13, 15, 24] to detect errors in the early stages of software development. However, they use lightweight static or dynamic analysis techniques that often make too simplistic assumptions or employ imprecise heuristics to analyze the source code, resulting in many false positives. Using the words of Ross [26], software development became more of a craft than a science. In fact, the theoretical work on computer science has presented multiple mature techniques to formally address such problems in software development, called formal methods.

Formal methods refer to the use of mathematical modeling and calculation in the specification, analysis, and assurance of software programs. However, such techniques cannot be applied directly because current programming languages do not come with a defined semantics that specifies the behavior of the resulting execution of the program. In fact, the lack of semantics can create an ambiguity between what a programmer writes and what a machine executes [16]. An example is the timestamps in the Java programming language that allows developers to assign timestamps integer values. This can lead to a runtime exception in the Java virtual machine when a timestamp with a negative value is used.

Recent works try to close this semantics gap by providing techniques [11] or compilers [19] which have a semantics that specifies how the source code is interpreted. Based on a defined time semantics, the objective of this thesis is to provide a verification framework to automatically **verify** and **extract models** of the time behavior of a program. We narrow the scope to the *time domain*, because with the recent growth of distributed and multi-threaded applications, the handling of time has become an important aspect. In fact, those applications manage time for different operations, such as limiting the waiting time for certain events or scheduling the execution of certain actions. Our research is based on the hypothesis:

Given a semantics of time we can reverse engineer and verify the time behavior in the execution of computer programs.

We use the following four main research questions to investigate our main hypothesis:

RQ1: How to define the semantics of time for programming languages?
RQ2: To what extent can we extract models of the time behavior from a given program?
RQ3: To what extent can we automatically identify errors in the time behavior of programs?
RQ4: To what extent can we verify the correctness of a program's time behavior?

The goal of this thesis is to provide developers with a set of methods and techniques to automate the verification of the time behavior of their programs based on a defined semantics of time. We design our research questions to achieve this goal incrementally. With the first research question we aim to define a time semantics for a programming language. With the second research question, we plan to extract a formal model of the time behavior in programs. The third research question will investigate means to automatically verify the observance of common predefined time properties in the source code. Finally, the last research question is designed to verify the time behavior not in isolation, but considering the full program specification. We decided to use the Java programming language to show the applicability of our approaches. However, all our findings can be generalized to other programming languages by adapting the implementation and the time semantics to the new language to support.

2 RESEARCH APPROACH

We will answer our research questions following the design science research methodology [29]. We plan to develop several prototypes that implement our approaches. The prototypes will be used to evaluate our approaches following the research methods described in [6]. We plan to use quantitative methods on open source Java projects because they provide a large corpus of source code that is publicly available. We will also use qualitative methods, such as surveys or semi-structured interviews, to collect the feedback of developers. The following sections describe each research question an how we plan to address them in detail.

2.1 Identification of Time (RQ1)

We plan to study the Java programming language and identify the mechanisms it offers to handle time. Based on this study, we will provide a definition of a formal time semantics for the language. A similar work on this aspect was performed by Bogdanas and Rosü [3]. They present a complete semantics for the Java programming language version 1.4 based on the K-Framework [27]. However, they follow the specification of Java and therefore their general semantics maintains some errors due to the Java representation of timestamps with integer values.

We addressed this problem in our previous research presented in [20]. We analyzed several packages of the Java standard library and identified two different ways offered by the Java to represent time. The first way is to use specific API classes that explicitly represent time, such as the `java.time.Clock` class. These classes enforce a specific behavior and semantic of time. The second way is to use an implicit representation of time using timestamps with integer values from the domain of $\mathbb{Z}$. In this research question, we focus on the second representation because it has no formally defined semantics. In fact, timestamps are considered merely an integer number without discerning the more strict properties of timestamps.

Based on this, we defined a formal time semantics of the language to cover the timestamp notation. Moreover, we defined an approach with the Rewriting Logic framework [21] to collect information from the source code of a Java class regarding the variables that store timestamps, called time variables. We evaluated our approach on 10 open source Java projects. The evaluation was used to verify the ability of our approach to identify time variables in terms of precision and recall. Our experiments show that we can identify time variables with a precision of 98.62% and recall of 95.37%.

2.2 Modeling the Time Behavior (RQ2)

The goal of this research question is to develop an approach to reverse engineering formal models of the time behavior from Java programs. These time models can be used to document the time behavior of existing programs and to verify properties of the software.

Modeling the time behavior of a program has been studied intensively over the last 20 years. A common technique to represent it is using timed automata, a formalism introduced by Alur [1], that allows to model and analyze timing behaviors. Many works use this representation to help developers monitor or analyze their systems. Jayaraman *et al.* [17] monitor the execution of a program verifying that its behavior conforms to a network of timed automata provided by the developers. Following a similar idea, Hakimipour *et al.* [9] and Georgiou *et al.* [8], show how to generate a program starting from a timed automaton. All of these recent techniques require the *manual* creation of such automata. Therefore, they are not suitable for existing programs from which we would like to reverse engineer the implemented time behavior. We plan to develop an approach to extract timed automata from a Java program at different levels of abstraction: method and class level.

Regarding the method level, we have developed a first approach presented in [20]. The extracted timed automata are amenable for verification using the UPPAAL [18] model checker. We evaluated our models showing their usability to discover 5 bugs collected from 4 different Apache projects. Moreover, we show that they can also be used to verify that the proposed patches can fix the five bugs.

Object-oriented languages, such as Java, use classes to encapsulate a specific behavior that is altered via method calls. Therefore, our next goal is to model the time behavior of a complete class. We plan to use Daikon [4] to identify likely invariants in Java classes. We will split the discovered invariants into two categories: data- and control-flow. Then, we plan to match those to their respective program points. Using our defined time semantics on the source code we can detect which invariants are time related. Once we have collected the time invariants of a class, we will define a framework based on Rewriting Logic [21] to create a timed automaton that model the time behavior of a full class.

We will evaluate our approach in terms of accuracy and usefulness. First, we will extract timed automata of a given real time program, and then, verify that the program behavior is aligned with the extracted one. The latter will be performed through monitoring the program during the execution of its test suite. Following the idea of Randoop [22–25], we plan to evaluate our extracted models also in terms of their ability to generate test cases. Since we have a model of the time behavior of the program, we can exploit it to produce a suite of regression tests to help developers to find out when a bug has been introduced.

2.3 Automatic Detection of Time Errors (RQ3)

With the third research question we plan to provide developers with an approach that can automatically detect time related errors in their source code. There are multiple tools that help developers in discovering problems in the source code. They can be divided into two categories: static analysis and test generation. The static analysis tools, such as FindBugs [15], perform a static analysis of the source code and, based on some predefine rules, they indicate which parts of the code may suffer from errors. The test generation tools analyze either statically (*i.e.*, Randoop) or dynamically (*i.e.*, Agitator [4]) an application, and then generate a test suite. The test suite can verify that some specific classes of the programming language are used correctly. It can also be used to automatically generate a regression test suite to warn developers when changes to source code have introduced an error. However, both techniques fail to detect time specific errors since they lack the knowledge of the time semantics.

With the first research question, we discovered that the time representation with timestamps using integer variables is error prone. Integer variables can also represent negative values that are not allowed for timestamps. In this research, we plan to use our proposed time semantics to identify the variables that store timestamp values. Based on this information, we plan to develop an approach that analyzes the source code and creates a slice of the original program. This slice will contain only those statements that use or alter the values of the identified time variables. Afterwards, we plan to translate the slice into an SMT model. The SMT model will also contain the time semantics and will represent the time behavior of the program that will be fed into an SMT solver. The SMT solver will be the oracle that reports all cases in which time properties cannot be held in the given model. The SMT solver will also report which constraints cannot be satisfied. Using the output of the SMT solver, our approach will report the statements in the code, that in some cases could alter time variables storing improper values, to the developers.

We plan to evaluate our approach qualitatively and quantitatively. With the qualitative evaluation we aim to find the theoretical limitation of our approach in terms of soundness and completeness. We expect the approach to be sound, *i.e.*, every detected error is a true error according to our time semantics, but not complete, *i.e.*, it may not identify all possible time related errors. The quantitative evaluation will verify how many errors our approach can identify in a dataset comprised of open source Java projects and how much time is required to compute the results. With this evaluation, we aim to discover if our proposed approach can be effectively used by developers to automatically discover time related errors at method level.

2.4 Formal Verification of Time in Programs (RQ4)

The goal of this research question is to semantically address the time domain at the level of full programs. Moreover, we will extend our semantics adding the semantics for the Java API classes that explicitly represent time.

With this research question we aim to introduce our extended time semantics into frameworks that consider the full semantics of the programming language. This will allow developers to prove time properties in the context of a full program. For this, we plan to extend two state-of-the-art frameworks: K-Framework and Java Path Finder (JPF). K-Framework is a rewrite-based executable semantic framework on which Bogdănaş and Roşu defined a formal semantics of the Java language version 1.4 [3]. JPF is a virtual machine for Java bytecode that can be used to verify and check properties of applications [12]. Both frameworks implement the specific execution semantics of Java. However, they do not provide any semantics for the time domain.

We will evaluate our approach using quantitative and qualitative studies. For the quantitative studies, we first plan to compare the two mentioned frameworks in terms of run time performance. Furthermore, we will evaluate the two frameworks and our extension on several open source Java projects collecting reports about the detected errors in the usage of time. We will manually verify each error to assess the accuracy and usefulness of our approach.

For every detected error, we will create a corresponding bug report in the issue tracker of the respective project. Through this, we aim at collecting the feedback of developers. In addition, we will send them short questionnaire or try to preform a semi-structured interview with them to evaluate the usefulness of our approach. The expected contribution of this research is an improvement of existing frameworks to support a richer semantics of the programming language that allows developers to verify a higher spectrum of properties and identify time related errors.

3 CONTRIBUTIONS

The expected contributions of this thesis are:

- A time semantics of the time domain for the Java programming language;
- An approach to extract timed automata at method and class level;
- A prototype tool to automatically detect time errors for timestamp variables;
- An improvement and extension of existing frameworks to allow developers to verify the time specific behavior in programs;

We have already published a study in which we presented a first definition of the time semantics for the Java programming language [20]. We showed its usability extracting timed automata from Java programs at the method level. Regarding our future work, we expect several implications of our studies for both, researchers and developers. For researchers, we will provide tools and a time semantics that can be used to tackle a new spectrum of problems that involve time, *e.g.*, the development of a certified compiler for the Java programming language. Furthermore, we will make our resources and prototype tools publicly available to allow other researchers to use and extend our studies. Concerning developers, we will provide tools that support the verification process of their programs. We expect that our contributions enable software developers to approach formal verification techniques with less effort. Furthermore, we expect that our methods and techniques to extract timed automata from source code can be used by developers to support the documentation of their programs.

4 RELATED WORK

In the following, we outline related work on timed automata, formal verification in software engineering, and formal semantics in programming language.

Timed Automata are a well known formalism to verify time properties of systems. Yang *et al.* [30] translate Simulink Stateflow models into UPPAAL timed automata. With UPPAAL they show the possibility to detect errors in the stateflows that could not be detected by Simulink. Jayaraman *et al.* [17] present an approach to monitor the execution of a program verifying that it conforms to a network of timed automata which specify the correct behavior of the program. The network of timed automata, however, needs to be created manually by the developers. Hakimipour *et al.* [9] propose a technique to translate timed automata into source code for real time applications. Similarly, Georgiou *et al.* [8] present an approach to create distributed programs from a network of input/output timed automata.

Formal Verification of source code was studied by researches at NASA who presented Java Pathfinder [12]. It is a framework for the verification of Java bytecode with a focus on detecting race conditions. Similarly, Bandera [10] is an approach to extract state machines that are amenable for formal verification. Another seminal work is presented by Henzinger *et al.* [14] to verify the correct usage of the mutex API in C programs.

Several **Formal Semantics** for many programming languages have been proposed. Regarding Java, three semantics are especially relevant for our work: ASM-Java [28], JavaFAN [7], and K-Java [3]. ASM-Java gives a formal semantics for the programming language, the compiler, and the bytecode representation of a program. JavaFAN compiles a subset of Java and its bytecode representation into the Maude language that allows model checking of LTL properties. The most recent semantic for the Java programming language is K-Java. They present a complete semantics for the programing language and the bytecode interpretation performed by the Java Virtual Machine. However, all three works lack a definition of the time semantics.

In our research, we close this gap and show that having such a semantics allows developers to formally verify time properties of their programs.

5 SCHEDULE

	1st Year				2nd Year				3rd Year			
RQ1	🟩	🟩	🟩									
RQ2		🟩	🟩	🟩			🟩	🟩				
RQ3				🟩	🟩	🟩						
RQ4								🟩	🟩	🟩	🟩	
Thesis Writing											🟩	🟩

In the first year, we developed the time semantics that formed the starting point of this thesis. Meanwhile, we also started to apply the time semantics to reverse engineering timed automata from Java source code at method level. This work was published at IEEE SCAM'17 [20] and it has been invited for an extension in a special issue of the EMSE Springer journal. During the second year, we plan to extend this work and extract timed automata at class level. In this time span, we will also focus on providing a technique to automatically detect time errors. We have already performed some work on the third research question that we submitted to IEEE SANER'18. The last year will be dedicated in extending existing works to support our defined time semantics to completely model check a program. Moreover, the last months of the third year will be dedicated to finalize the thesis.

REFERENCES

[1] Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.

[2] C Artho. 2006. JLint – Find Bugs in Java Programs. (2006). http://jlint.sourceforge.net/.

[3] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 445–456.

[4] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA)*. ACM, 169–180.

[5] Robert N Charette. 2005. Why software fails. *IEEE spectrum* 42, 9 (2005), 1–36.

[6] John W Creswell. 2013. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

[7] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. 2004. Formal analysis of Java programs in JavaFAN. In *International Conference on Computer Aided Verification*. Springer, 501–505.

[8] Chryssis Georgiou, Peter M Musial, and Christos Ploutarchou. 2013. Tempo-toolkit: Tempo to Java Translation Module. In *International Symposium on Network Computing and Applications (NCA)*. IEEE, 235–242.

[9] Niusha Hakimipour, Paul Strooper, and Andy Wellings. 2010. TART: Timed-automata to real-time Java tool. In *International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 299–309.

[10] John Hatcliff and Matthew Dwyer. 2001. Using the Bandera tool set to model-check properties of concurrent Java software. In *International Conference on Concurrency Theory (CONCUR)*. Springer, 39–58.

[11] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345.

[12] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 366–381.

[13] Nikolas Havrikov, Alessio Gambi, Andreas Zeller, Andrea Arcuri, and Juan Pablo Galeotti. 2017. Generating unit tests with structured system interactions. In *Proceedings of the 12th International Workshop on Automation of Software Testing (AST)*. IEEE Press, 30–33.

[14] Thomas A Henzinger, George C Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. 2002. Temporal-safety proofs for systems code. In *International Conference on Computer Aided Verification (CAV)*. Springer, 526–538.

[15] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM Sigplan Notices* 39, 12 (2004), 92–106.

[16] Michael Jackson. 1995. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*. ACM, 283–292.

[17] Swaminathan Jayaraman, Dinoop Hari, and Bharat Jayaraman. 2015. Consistency of Java run-time behavior with design-time specifications. In *International Conference on Contemporary Computing (IC3)*. IEEE, 548–554.

[18] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)* 1, 1-2 (1997), 134–152.

[19] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. The CompCert memory model. In *Program Logics for Certified Compilers*, Andrew W. Appel (Ed.). Cambridge University Press.

[20] Giovanni Liva, Muhammad Taimoor Khan, and Martin Pinzger. 2017. Extracting timed automata from Java methods. In *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 91–100.

[21] José Meseguer and Grigore Roşu. 2011. The rewriting logic semantics project: A progress report. In *International Symposium on Fundamentals of Computation Theory*. Springer, 1–37.

[22] Carlos Pacheco and Michael D Ernst. 2005. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 504–527.

[23] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA)*. ACM, 815–816.

[24] Carlos Pacheco, Shuvendu K Lahiri, and Thomas Ball. 2008. Finding errors in. net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA)*. ACM, 87–96.

[25] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*. IEEE Computer Society, 75–84.

[26] Philip E Ross. 2005. The exterminators [software bugs]. *IEEE spectrum* 42, 9 (2005), 36–41.

[27] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[28] Robert F Stärk, Joachim Schmid, and Egon Börger. 2012. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media.

[29] R Hevner Von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram. 2004. Design science in information systems research. *MIS quarterly* 28, 1 (2004), 75–105.

[30] Yixiao Yang, Yu Jiang, Ming Gu, and Jiaguang Sun. 2016. Verifying simulink stateflow model: timed automata approach. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 852–857.