NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
“COMPUTER SYSTEMS: SOFTWARE AND HARDWARE”

MASTER THESIS

Datalog Based Symbolic Program Reasoning for Java

Christos V. Vrachas

Supervisor: Yannis Smaragdakis, Professor NKUA

ATHENS

June 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
“ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμβολική Συλλογιστική Προγραμμάτων για Java, βασισμένη σε Datalog

Χρίστος Β. Βραχάς

Επιβλέπων: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

Ιούνιος 2019

MASTER THESIS

Datalog Based Symbolic Program Reasoning for Java

Christos V. Vrachas **R.N.:** M1608

SUPERVISOR: **Yannis Smaragdakis,** Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμβολική Συλλογιστική Προγραμμάτων για Java, βασισμένη σε Datalog

Χρίστος Β. Βραχάς **Α.Μ.:** M1608

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ABSTRACT

abstract english

SUBJECT AREA: Static Program Analysis, Symbolic Reasoning, Propositional Logic

KEYWORDS: flow-sensitivity, path-sensitivity, inference rules and logic proofs

ΠΕΡΙΛΗΨΗ

abstract greek

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων, Συμβολική Συλλογιστική, Προτασιακή Λογική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ”ευαισθησία”-ροής, ”ευαισθησία”-μονοπατιού, κανόνες συμπερασμού και αποδείξεις προτασιακής λογικής

This master thesis is dedicated to ...

ACKNOWLEDGEMENTS

I would like to thank ....

June 2019

CONTENTS

1 Introduction **1**

1.1 Thesis Structure 1

2 Background **2**

2.1 Static Program Analysis 2

2.1.1 Whole-Program vs. Modular Analyses 2

2.1.2 Flow-Sensitivity 3

2.1.3 Path-Sensitivity 3

2.2 Symbolic Execution 4

2.3 Theorem Provers 4

2.4 Datalog 5

3 Static Declarative Symbolic Reasoning **6**

3.1 Schema Relations and Program Expression Trees 6

3.2 Path Expressions 6

3.3 Boolean Symbolic Reasoning 6

3.3.1 Propositional Logic Axioms 6

3.3.2 Propositional Logic Inference Rules 6

4 Evaluations **7**

5 Related Work **8**

6 Conclusions and Future Work **9**

Abbreviations - Acronyms **10**

Appendices **10**

REFERENCES **11**

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SOURCE CODES

1. INTRODUCTION

FooBar

1.1 Thesis Structure

BarBar

2. BACKGROUND

Several program analysis techniques have been proposed in the literature, in order to aid developers and users discover interesting program properties within their software. For example, one might be interested in finding out whether there are memory leaks or whether some piece of code is reachable.

Such techniques come in different flavors, either static or dynamic, and are usually based on strong mathematical concepts. In this section we provide a brief background on these concepts and the frameworks utilized towards the development of program analysis tools.

2.1 Static Program Analysis

Static program analysis is a program analysis technique that aims to reason about a program's behaviors without actually executing it. It has been heavily utilized by optimizing compilers since their early stages and it has also found several other applications, among the areas of softare security, software correctness [9]. The question of whether a program is correct or may terminate for all possible inputs is in general undecidable. However, static analysis techniques are able to tackle undecidability by overapproximating or underapproximating the initial problem, attempting to reason over a simplified version of it.

There is a plethora of different static analysis algorithms in the literature, each one met in several domains. For instance, one may utilize analyses such as *liveness* and/or *pointer* analyses in an optimizing compiler with the intention of eliminating dead code regions or performing a constant propagation/folding optimization. Similarly, a *reachability* analysis that determines whether a specific program point is reachable could be used by a software correctness tool to make sure that an erroneous state is actually never reached.

There is a variety of design choices that may prove essential towards the *scalability* and *precision* of a static program analysis algorithm. We briefly describe some of those choices in the context of this work.

2.1.1 Whole-Program vs. Modular Analyses

Whole-Program analyses need not be confused with *inter-procedural* analyses. An inter-procedural analysis considers multiple functions in order to perform its computations. However, a *whole*-program analysis does not only reason about a program's properties across different functions, but also includes any external dependencies during its computations, such as external libraries or native calls.

On the contrary, a *modular*-analysis reasons about specific parts of a program, disregarding dependencies across different calls and external code. Modular-analysis is fundamentally proportional to *intra*-procedural analyses. In the intra-procedural setting, an analysis would restrict its reasoning within a specific function bound, overlooking the way that function calls or external dependencies affect the computations inside the function. However, someone could also perform a modular-style analysis across a relatively small set of functions, that is an *inter-procedural* analysis.

Whether the analysis would reason about a program's properties in a whole-program or

a modular manner is of great importance, because such a decision affects the way that the analysis practitioner would tune its performance - scalability and precision. In the whole-program analysis setting the design tradeoffs between scalability and precision might prove crucial towards the overall analysis reasoning, though a modular-analysis may be able to perform more precise reasoning, due to the relatively restricted area of interest. Whole-program analyses are commonly used in the analysis of languages with complex language features. Those languages, such as the object-oriented ones need to reason about language constructs that may reside on the heap, and thus they may benefit from a whole-program analysis reasoning. Such an analysis may not take into consideration the exact ordering of the instructions of a program, sacrificing precision towards better scalability.

2.1.2 Flow-Sensitivity

The concept of whether an analysis is designed with respect to the instruction ordering is called *flow-sensitivity*. On the contrary, an analysis that is not designed that way is called flow-insensitive. Flow-sensitivity is tightly related with the scalability and precision of static program analyses, due to the fact that a flow-sensitive analysis keeps track of program properties at every point of it, eg. before or after every instruction. Whole-program analyses may usually omit the control-flow constructs during any of their reasoning. However, many whole-program static analyses like the Doop framework's Pointer analysis for Java manage to add flow-sensitivity to their core analysis by slight preprocessing [10].

Several tools utilize state-of-the-art compiler technologies to convert the input source to a lower level *intermediate-representation (IR)*. For instance, there is a plethora of tools that utilize the *LLVM Compiler Infrastructure* [8] for *C*-like languages, or *Soot - A Java optimization framework* [11] - for JVM languages. These tools may further lower the source code in a *Static Single Assignment (SSA)* form. In SSA form, each variable is assigned exactly once, never to be re-assigned again and it is also defined before any of its uses, thus yielding a flow-sensitive representation.

2.1.3 Path-Sensitivity

A path-sensitive analysis is in a way an enhanced version of a flow-sensitive analysis which also considers the path taken up to a specific program point. Such an analysis introduces a path representation usually encoded as the combination of a program's neighboring branch conditional expressions, named *path predicate*. A path predicate is essentially a *Boolean formula*, that is a function of boolean variables whose assignment yields a different control-flow path.

Explicit knowledge of the path taken up to a specific program point allows the analysis to achieve higher precision, ideally eliminating the need for any approximation. However, such knowledge comes at a cost; the number of a program's paths is exponential to the number of branches within the program, thus an analysis that keeps track of all possible paths would not manage to scale. There have been suggested multiple techniques [6], [3] to address such issues in the literature. Some of those techniques manage to achieve scalability by restricting the context of the computations or by introducing loose abstractions of the initial problem. As a static analysis technique a path-sensitive analysis does not eventually execute the program, rather utilizes the path encoding in order to assert properties that hold at specific program points.

One of the adverse effects of pure Static Analysis techniques is that they result a large amount of *false positives* by trading off precision for scalability. In the context of such techniques, a false positive is said to be an erroneous report of a static analysis tool that a property violation has been discovered within the analyzed program, though such violation does not eventually exist. There have been proposed several techniques of either (semi-) dynamic nature that execute directly the program via code instrumentation, or static solutions that attempt to simulate the execution of a program.

2.2 Symbolic Execution

Symbolic Execution is yet another program analysis technique that may be of either static or dynamic flavor and mainly attempts to answer whether certain properties of a program could potentially be violated by a piece of code [1]. At the same time it also results an automated way to generate test cases for programs under testing.

In contrast to the aforementioned static analysis techniques that do not directly execute a program, Symbolic (or *Concolic*) Execution mainly attempts to simulate the execution of a program by considering symbolic (non-deterministic) values for its input. Usually, during the concrete execution of a program a single execution path is cosidered for any computation, whilst a Symbolic Execution engine manages to explore a plethora of paths thanks to the symbolic values assigned to its input variables. As such, a concrete execution is said to under-approximate the desired analysis. *Concolic Execution* is a mixture of symbolic and concrete execution that considers concrete input values when possible, thus making symbolic execution feasible in practice. The latter is usually described as Dynamic Symbolic Execution (DSE). Several tools have been proposed in the literature such as SAGE [7] and KLEE [2], that are considered to be the tools of choice when it comes to binary analysis and/or systems testing.

In the symbolic setting, execution is typically driven by a *symbolic execution engine* which at its core maintains some state for every explored path. The core data-structures of such an engine are the *Boolean formulas* that encode an execution path by considering the satisfied conditions taken for that exact path, and a *symbolic store* that keeps a mapping between the variables met within the path and their equivalent symbolic expressions or values. The former are constructed on each branch execution, while the latter is updated after an assignment to the corresponding variable. Established by those two structures for a path, the engine utilizes an *automated theorem prover* whose purpose is to check whether there is any violation of the property under analysis that would lead to a failed execution, besides verifying path feasibility. A path is said to be feasible in such scenario, if there exists an assignment of concrete values to its symbolic encoding that would yield the satisfiability of its boolean formula. Along with *path-explosion* due to the exponential state space exploration, non-deterministic inputs and several other vulnerabilities, symbolic execution efficiency also relies to the use of efficient external theorem provers. Ongoing research attempts to face those threats in order to provide better tools that aid developers and help discover bugs in programs.

2.3 Theorem Provers

Automated theorem proving, namely the ability to automatically prove a mathematical theorem has been the insatiable desire of the scientific community for an extensive period

of time. Theorem proving's establishment is essentially mathematical logic formulas and modelling. On the other hand, *Computer Science* foundations lie within mathematics and logic. Many times software developers want to prove that a specific property is satisfied for a program and its given input values, like its termination. Computer programs can also be encoded as a logical formula and so formal theorems may be constructed in the programming setting too.

The purpose of an automated theorem prover (ATP) is ideally to answer whether a theorem can be proven or not, providing at the same time a formal proof or a counterexample. Automated theorem proving was initially considered to be part of Artificial Intelligence (AI). However, human assistance would be essential in many cases, leading at he same time to the recognition of unable to be proven theorems (such as termination).

Automated theorem proving has lead to the creation of several tools following different approaches to solve the same problem: proof construction. Of particular interest have been satisfiability modulo theories (SMT) provers (or solvers) that are the main tools employed in symbolic execution. SMT is a generalization of the traditional boolean satisfiability problem (SAT) that utilizes underlying decidable mathematical theories in order to encode and give semantical meaning to function predicates instead of boolean (binary) variables. Such theories are the theory of linear arithmetic, bitvectors and arrays and provide a powerful expressiveness to the related solvers which could also be considered as constraint satisfaction engines. Provided a predicate formula encoding, a prover aims to check whether the formula is satisfiable; there is an assignment of values to the compounded predicates. The state of the art tools for theorem proving have been the Z3 Theorem Prover [4], and the Coq Proof Assistant [5], an interactive theorem prover

2.4 Datalog

// TODO: Datalog example, popular Datalog use cases and applications in Static Analysis

3. STATIC DECLARATIVE SYMBOLIC REASONING

our approach

3.1 Schema Relations and Program Expression Trees

input facts and Reasoning

3.2 Path Expressions

pathhz

3.3 Boolean Symbolic Reasoning

reasoning logic

3.3.1 Propositional Logic Axioms

pure axioms

3.3.2 Propositional Logic Inference Rules

inferencer

4. EVALUATIONS

foolala

5. RELATED WORK

foorelated

6. CONCLUSIONS AND FUTURE WORK

FooConclusions

ABBREVIATIONS - ACRONYMS

IR	Intermediate Representation
JVM	Java Virtual Machine

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[3] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, volume 37, pages 57–68. ACM, 2002.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[6] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[10] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.