Aalto University
School of Science and Technology
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

Mikko Lahola

Flow Analysis Directed Symbolic Execution for Model-Based Test Generation

Master's Thesis

Espoo, April 29, 2010

Supervisor: Professor Ilkka Niemelä
Instructor: Antti Huima, M.Sc.(Tech.)

Aalto University
School of Science and Technology

Aalto University School of Science and Technology Faculty of Information and Natural Sciences Degree Programme of Computer Science and Engineering	ABSTRACT OF MASTER'S THESIS

Author	Mikko Lahola				
Title of thesis	Flow Analysis Directed Symbolic Execution for Model-Based Test Generation				
Pages	x + 57	Date	April 29, 2010	Language	English
Professorship	Theoretical Computer Science			Code	T-119
Supervisor	Professor Ilkka Niemelä				
Instructor	Antti Huima, M.Sc.(Tech.)				

Formal program analysis methods have been used to aid test case generation for software testing for some time now due to their ability to generate tests with less labor and of better quality than those done by manual means. However, the performance of the methods is commonly hampered by state space explosion.

In the thesis the focus is on improving the performance of an industrial-strength tool that generates tests in a model-based testing setting. The tests are generated with a program analysis method called symbolic execution that finds execution paths to syntactic control flow points of the model. The industrial setting requires that the tool can handle infinite-state models, and to this effect, the execution path lengths are bounded in the symbolic execution system.

The thesis presents a technique to optimize the execution path search performed by the symbolic execution system. The optimization is based on the idea that only the execution paths that reach new syntactic control flow points within the given bound are relevant to the search. To identify the relevant paths, a structure containing a safe overapproximation of the possible execution paths of the model is computed, and during symbolic execution path lengths in the structure are compared to the search bound. The structure is computed with a common compiler technique called flow analysis.

Also, as the structure is chosen to be equivalent to a pushdown automaton the thesis presents an algorithm to calculate shortest paths in pushdown automata. The impact of the search optimization technique and the efficiency of the shortest path calculation algorithm are empirically quantified with industrial models.

Keywords

symbolic execution, flow analysis, model-based software testing, pushdown automata

Aalto-yliopisto
Teknillinen korkeakoulu

Aalto Yliopisto Teknillinen korkeakoulu Informaatio- ja luonnontieteiden tiedekunta Tietotekniikan koulutusohjelma			DIPLOMITYÖN TIIVISTELMÄ
Tekijä	Mikko Lahola		
Työn nimi	Mallipohjainen testigenerointi vuoanalyysin ohjaamalla symbolisella suorittimella		
Sivumäärä	x + 57	Päiväys 29. huhtikuuta 2010	Julkaisukieli englanti
Professuuri	Tietojenkäsittelyteoria		Koodi T-119
Työn valvoja	Professori Ilkka Niemelä		
Työn ohjaaja	DI Antti Huima		

Ohjelmistotestauksessa käytettyjä testitapauksia on tuotettu käyttäen formaaleja ohjelma-analyysimenetelmiä. Menetelmien etuna on, että testit voidaan tuottaa vähemmällä työmäärällä saaden niistä kuitenkin käsintehtyjä parempilaatuisia. Menetelmien yleisenä tehokkuusongelmana on kuitenkin analysoitavien ohjelmien tila-avaruuden valtava koko.

Työssä tarkastellaan mallipohjaisessa testauksessa käytetyn testitapauksia tuottavan teollisen ohjelmiston tehostamista. Ohjelmisto käyttää symbolinen suoritus -tyyppistä ohjelma-analyysimenetelmää etsiäkseen suorituspolun mallin jokaiseen syntaktiseen kontrollivuon pisteeseen. Teollisuuskäytöstä johtuen ohjelmiston on käsiteltävä äärettömän tila-avaruuden malleja ja siksi ohjelmiston symbolisessa suorittimessa on asetettu raja käsiteltävien suorituspolkujen pituudelle.

Työssä esitetään symbolisen suorituksen tekemää suorituspolkujen etsintää optimoiva tekniikka. Ideana on, että vain ne suorituspolut, jotka saavuttavat uusia syntaktisia kontrollivuon pisteitä annetun rajan puitteissa, ovat oleellisia. Oleellisten polkujen tunnistamiseksi lasketaan rakenne, joka esittää turvallista yliapproksimaatiota mallin mahdollisille suorituspoluille ja verrataan suoritusaikaisesti rakenteesta löytyvien suorituspolkujen pituuksia annettuun haun rajaan. Rakenne tuotetaan kääntäjätekniikassa yleisesti käytetyllä vuoanalyysimenetelmällä.

Työssä esitetään myös algoritmi vuoanalyysin tuottaman pinoautomaattirakenteen minimipolkulaskennalle. Esitellyn optimointitekniikan vaikutusta ohjelmiston symbolisen suorittimen tehokkuuteen ja minimipolkulaskennan tehokkuutta arvioidaan kokeellisesti teollisuusesimerkeillä.

Avainsanat	symbolinen suoritus, vuoanalyysi, mallipohjainen ohjelmistotestaus, pinoautomaatit

Acknowledgements

Working on the thesis has been an arduous process. Luckily, I have had several people helping me during this time and many thanks are in order. First I want to thank my instructor Antti for introducing the general topic of the thesis to me and for the expert views in technical matters. I want to thank my supervisor Ilkka for consistently finding the time to review the thesis for structure and content in the midst of a ridiculously busy schedule. I want to thank Conformiq Software for the financial support and my colleagues there for their insightful comments that eased the process significantly. I want to thank my parents Raimo and Mervi and my sister Katri, as well as all my friends for the mental support and for putting up with me even as I became increasingly irate during the process. Thank you Maija for being there for me.

Espoo, April 29, 2010

Mikko Lahola

Contents

Abbreviations

CFG	control flow graph
DAG	directed acyclic graph
DFA	data flow analysis
k-CFA	k-order control flow analysis
PDA	pushdown automaton / pushdown automata
SUT	system-under-test

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This master thesis is a part of the ongoing work at Conformiq Software Inc. in developing methods for automated software testing. The goal of the thesis is to explore the feasibility of reducing the automated test generation time of Conformiq's flagship product Qtronic [1] by finding and cutting unnecessary parts of the test generation algorithm's search space. The focus is on real-world software testing scenarios (infinite-state systems).

1.1 Background

Software testing is recognized as an important part of the software engineering process [48]. There are many industrial examples of inadequate testing leading into critical systems failing and causing big personal and monetary losses. Classical examples of costly software bugs include the floating-point division bug in the Intel Pentium processor leading into a mass recall of chips from the market [47], the failed launch of the Ariane 501 satellite due to a chain-reaction triggered by an unexpected input value [26] and a Therac-25 medical electron accelerator treating six patients to a radiation overdose due to (amongst other issues) a race condition in the equipment [41].

There are several ways of trying to ensure the creation of bug-free software systems. We use the term software testing to refer to the methodology of finding bugs in already created software as opposed to preventing the bugs from being created at all. We also consider the finding of bugs to happen via the usage of *test cases* [43] as opposed to directly pointing them out from software code.

Software testing accounts for the largest percentage of technical effort in a software engineering process [48]. Historically testing has been carried out mostly *manually* - a test engineer examines the system-under-test (SUT), laboriously writes out test cases, runs them against the system and writes down findings. Like in other information processing fields this process has been gradually *automated*. Nowadays, many companies find use of automated testing in the sense that the testing infrastructure - test running, reporting, etc. - is automated but the test cases are still written by hand, or generated from templates written by hand. [28]

Automated testing can be done in many different ways. We differentiate methods by three properties [43]:

1. The software can be tested by running the actual software against specific cases (*dynamic testing* or *dynamic analysis* of the software) or by not running the software but instead analyzing source code or similar (*static testing* or *static analysis*). These approaches can be adjoined in the way that static analysis is used to generate test cases that are later run in a dynamic analysis fashion.
2. A differentiation can be based on whether the method has knowledge of the internal structure and implementation details of the software (*white-box testing*) or whether it only accesses the external interface of the software, such as inputs and outputs (*black-box testing*).
3. We can look for direct programming errors in the software (usage of uninitialized data, division by zero, etc.) or create tests against functional requirements (e.g. a certain input will lead to a certain sequence of outputs), the latter being referred to as *functional testing*.

In this thesis the environment in which testing is performed is *model-based testing*, an approach in which test cases are generated from a model of the software. A *model* is created separately from the software that is to be built and contains an abstracted view of the desired functionality of the software - it is a mathematically structured form of functional requirements for the software [23]. The test generation process can be viewed as a form of static functional testing where we have access to the inner details of the model of the software (white-box) but the actual software is seen as a black-box. From now on in the thesis, we use "model" to always refer to the model of the software that is to be tested, i.e. the model of the system-under-test (SUT).

Automated test case generation methods - in particular those based on *formal methods* [18] - are transferring from academia to industrial applications. The case for the methods is that (a) they reduce the required amount of labor in creating tests [28], and - perhaps more importantly - (b) they have been seen to generate tests of better quality i.e. tests that cover systems with complex functionality better [29].

Automated test case generation methods for model-based testing share a common problem: the number of computational states in a model grows exponentially in the number of some structural features, such as the amount of variables used in the model. This is called the *state explosion problem* and it is a subject of ongoing research with no immediate end in sight. [59, 16]

Conformiq Software produces an automated model-based test generation tool called Qtronic [1]. The models given to the tool are written in Java [31] and UML [45] and tests are generated by finding concrete execution paths from the initial state of the model to those syntactic control flow points of the model (e.g. code lines) that are requested by the user of the tool. This means that the user can dictate the required coverage, which could be e.g. *branch coverage* (where each branch would be a target for the search) or *method coverage* (where the starting point of each method would be a target). We focus on the default coverage criteria of the tool, which includes both branch and method coverage as well as states and transitions of the UML diagram.

Before test generation the models are compiled into a functional language that is very similar to Scheme [2]. The resulting Scheme-like model is instrumented so that there are markings on the syntactic control flow points that correspond to those syntactic control flow points of the original model that were specified by the user. To generate tests, an in-house implementation of a program analysis method called symbolic execution is used. The search goal of the symbolic execution system is to cover the marked control flow points of the Scheme-like model by finding concrete execution paths from the initial state of the model to each marked point. The tool is not exempt from state explosion and the efficiency of the test generation algorithm is one of the main development issues of the tool.

Symbolic execution - with its origins already in the 1970s [11, 38, 19, 35] but with continued interest in academia [49] as well as tool support [14, 13] - is a current and popular approach to handle state explosion. In symbolic execution variables of the program under analysis are treated symbolically allowing them to represent multiple actual values at once. This enables a single symbolic execution path to cover multiple actual execution paths and therefore reduce the amount of states to analyze leading into faster analysis systems.

Despite its benefits over concrete execution, model-based test generation via symbolic execution suffers from the following (as-of-yet) unsolved problems:

1. The execution time of a symbolic execution algorithm is generally at least exponential to the size of the input model. This is due to the time taken to find actual possible execution paths out of symbolic execution paths - usually done with a decision procedure such as a constraint solving system [6].
2. Applying symbolic execution to models with infinite execution paths requires limiting the analysis in some way to make the analysis finite. Symbolic execution is then only able to find an underapproximation of the actual semantics of the system, i.e. a subset of the possible execution paths.

Since Qtronic is targeted at analyzing real-world models, it has to cope with infinite execution paths. The limiting of its analysis is done by using a heuristic: the symbolic execution of a path is terminated if some amount of "certain type of states" are reached since the last new marked syntactic control flow point. The heuristic specifically allows infinite paths as long as they do not contain "certain type of states". The heuristic will be discussed in more detail during the thesis.

The search criterion in this kind of functional testing has an essential difference to other forms of testing. Whereas symbolic execution schemes that search for programming errors in code try to cover all possible execution paths to a given bound - going through the same syntactic control flow points as many times as necessary to do so - here we are only interested in covering each (marked) syntactic control flow point once. This allows for a possible optimization in search: there is no need for the symbolic execution system to expand search on paths that would not reach new (marked) syntactic control flow points within the given bound.

Example 1.1. Figure 1.1 shows a small procedural program. The possible execution paths of the program are given in graph-form in Figure 1.2, where the label on each

```
 1: if x == 1 then
 2:    PRINT "A"
 3: else
 4:    if y > 0 then
 5:       z := -1
 6:    else
 7:       PRINT "B"
 8:    end if
 9:    if z < 0 then
10:       PRINT "C"
11:    else
12:       PRINT "D"
13:    end if
14: end if
```

Figure 1.1: Example procedural program.

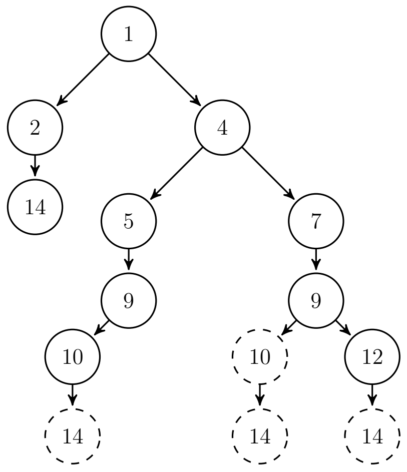

Figure 1.2: Execution paths of the program in Figure 1.1.

node corresponds to a syntactic control flow point (line number) of the program. (We assume that each line number is a target for the search here.) Now, assuming that the symbolic execution search always takes the "then" branch of the if-then-else construct first (corresponding to always taking the left child node first in the graph), then the execution paths corresponding to the dashed nodes in the graph are not necessary since they do not cover new syntactic control flow points. In particular, note that line number 9 has to be covered twice to find a path to all points corresponding to line numbers; determining where the search can be stopped is not straightforward. Another example with a model given in Java is discussed later on in the thesis.

The question is, how to identify the paths that do not reach new syntactic control flow points within the given bound? This thesis focuses on identifying a (hopefully large) subset of the paths with a static analysis method called *flow analysis*. Flow analysis produces a finite structure of the overapproximated control flow of the model under analysis, which gives an upper bound to the possible real execution paths of the model. Therefore, if we can, during symbolic execution, deduce from this precomputed structure (and from the information given by symbolic execution as it is run) that by continuing execution from a certain path we cannot reach new syntactic control flow points within the given bound the path can be cut from the search. Given the exponential-time nature of symbolic execution the hypothesis is that cutting the search in this way could reduce test generation time considerably, provided that the flow analysis system and the deduction can be implemented efficiently enough.

To summarize, we consider a real-world model-based testing environment with automated test case generation via a tool that runs symbolic execution on models given in a Scheme-like language. The symbolic execution search goal is to cover the given model by finding at least one execution path from the initial state of the model to each marked syntactic control flow point of the model that can be reached within the given bound. The main effort of the thesis is to give an answer to the following question: can symbolic execution time, and therefore test generation time, be substantially reduced by determining an upper bound to the possible execution paths of the given model with flow analysis, and, during symbolic execution, identifying (and cutting symbolic execution search on) some of the paths from which expansion of the symbolic execution search would not find any new marked syntactic control flow points within the given bound?

We realize beforehand that our potential design will not, in the worst case, be able to (positively) affect the running time of the symbolic execution algorithm since there can always be input models that by structural features alone do not allow cutting the search at any point, e.g. a model with a marked syntactic control flow point at the end of its single possible execution path (that can be reached within the given bound). Therefore we focus on giving an answer to the main problem by dividing the problem into three parts:

1. How substantial can the improvement in test generation times be in cases where the symbolic execution search is cut substantially?
2. How substantial can the induced overhead of our design be in cases where the symbolic execution search is not cut substantially?
3. Is it more common that our design is able or that it is not able to cut symbolic execution search substantially?

1.2 Related Work

There is a wide and ever-growing variety of symbolic execution and static analysis systems available. A recent survey on symbolic execution discusses some systems as well as gives guidelines on the popular research topics in the field [49]. Another survey on formal software verification [27] compares static analysis tools such as BLAST, SATABS and MAGIC. Symbolic execution has been used in conjunction with static analysis, e.g. in [4, 8] and as well as in static analysis articles, flow analysis is also discussed in compiler literature [3].

These systems are useful reference material to the implementation of our flow analysis system as well as for comparisons with the symbolic execution system used in Qtronic. However, we are specifically interested in systems that try to cut symbolic execution search in a manner similar to what was described in the last section. For this, we look for systems that (a) use bounded symbolic execution to generate tests, (b) have the search goal of finding at least one execution path to each designated point, and (c) use some heuristic to direct the search.

Other symbolic execution systems, such as the popular EXE [14] and its successor KLEE [13], can generate tests in a similar manner as our symbolic execution system and could be easily modified to have a bound on the search. However, they are often designed for directly finding bugs from program code and therefore do not really direct the search since the search goal is to cover each path of the program for completeness.

Model checking [17] is considered as a good method of verifying *safety properties* [27], i.e. properties which assert that "nothing bad happens". The method generates counterexample traces for violations of such properties and if we consider, somewhat perversely, that finding an execution path to a marked syntactic control flow point constitutes as "something bad" then the method is usable for finding the execution paths our test generation task requires. Model checking is a broad field and advances in the field can be applicable to symbolic execution systems [60]. As an example of existing tools we note two frameworks for which many tools have been implemented for, SPIN [34] and Java PathFinder (JPF) [12]. The bounded variation of model checking, called the *bounded model checking* [10] approach, is different from our symbolic execution system in the sense that the bound used in bounded model checking is usually some fixed amount of steps of execution whereas we allow for a more general bounding heuristic.

A recent trend in symbolic execution is to mix symbolic and concrete execution, which is referred to as *dynamic symbolic execution* or *concolic testing* [49]. The idea is to execute the system-under-test concretely but to at the same time collect symbolic information about the executed path. The symbolic information is then used to give input values for the concrete execution so that it will cover new execution paths. Tools include DART [30], CUTE [52] and PEX [58]. These systems do direct the search based on a heuristic, but usually the search goal is either to cover all paths, and therefore directing the search just gives partial results sooner (DART, CUTE), or the search is bounded so that only a given number of paths are executed through, in which case the heuristic dictates which partial results are found (PEX).

To our best knowledge there is no existing system that would direct bounded symbolic execution search with information from flow analysis, especially in the model-based test generation application field. However, with the large amount of program analysis tools produced over the years, we realize that there could be some systems that we overlooked.

1.3 Structure of the Thesis

The rest of the thesis is structured in the following way:

Chapter 2 contains the required background theory of graphs, pushdown automata, fixpoints and the two program analysis methods covered in the thesis: symbolic execution and flow analysis. Chapter 3 gives a closer look into model-based testing, in particular to the models that are used as inputs and the symbolic execution system used. Chapter 4 presents the main contribution of the thesis, the system which cuts symbolic execution search dynamically with flow analysis information. Chapter 5 focuses on the design and implementation of an efficient algorithm for calculating shortest paths in pushdown automata. Chapter 6 contains the empirical results obtained by testing our design on industrial models. Chapter 7 presents an analysis of the empirical results, discusses the applicability of the results and gives alternative approaches. Finally, Chapter 8 concludes the discussion with an eye towards future work on the subject.

Chapter 2

Background Theory

This chapter gives the mathematical definitions of graphs, pushdown automata and fixpoints that are required in the latter chapters of the thesis and discusses the theoretical basis of two program analysis methods: symbolic execution and flow analysis.

2.1 Mathematical Constructions

2.1.1 Graphs

We discuss graphs first. Graph theory is a vast field and we consider here only the parts of it necessary in the thesis, namely weighted directed acyclic graphs and the single-source shortest path calculation problem for them. For further details in graph theory see e.g. [25] and as an introduction to graph problems and algorithms see e.g. [20].

Definition 2.1 (Graphs). A *graph* $G = (V, E)$ is a set of nodes called *vertices* of the graph V and a set of *edges* $E \subset (V \times V)$ consisting of pairwise connections between vertices.

Definition 2.2 (Graph properties). A graph is *undirected* if $a, b \in V, (a, b) \in E$ implies $(b, a) \in E$. Conversely, if no such implication exists the graph is *directed*.

A *path* P between two vertices $x, y \in V$ of a (directed) graph is a sequence of edges $((v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n))$, where $\forall i \in \mathbb{Z}, 1 \leq i \leq n - 1 : (v_i, v_{i+1}) \in E$, so that $x = v_1$ and $y = v_n$. Alternatively, we write the path as a sequence of nodes. In this case such construction would be $P = (x, v_2, v_3, ..., v_{n-1}, y)$.

A directed graph contains a *cycle* if $v \in V$ and there exists a path $P = (v_1, v_2, ..., v_n)$ so that $v = v_1$ and $v = v_n$. A graph that contains a cycle is *cyclic*. Conversely, a graph is *acyclic* if it does not contain a cycle.

A graph is a directed acyclic graph (DAG) if it is both directed and acyclic.

Definition 2.3 (Weighted graphs). A *weighted graph* is a graph where each edge has an associated *weight*, typically given by a *weight function* $w : E \to \mathbb{R}$. We

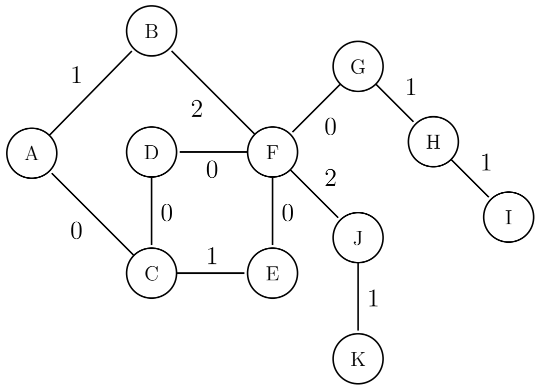

Figure 2.1: An undirected graph.

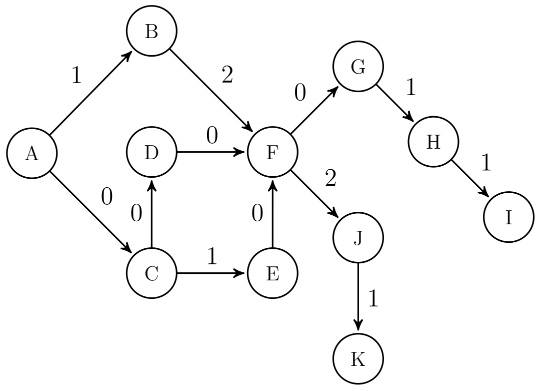

Figure 2.2: A directed acyclic graph.

constrain edge weights to be non-negative integers, giving $w : E \rightarrow (\mathbb{Z}_+ \cup \{0\})$. For a shorthand, we set $\mathbb{Z}_0 = (\mathbb{Z}_+ \cup \{0\})$. The weight of a path, or *path weight*, of $P = (v_1, v_2, ..., v_n)$ is the sum of the weights of its edges:

$$W(p) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

Definition 2.4 (Shortest path problem). Given $x, y \in V$, the *shortest path weight* from x to y is defined by:

$$\delta(x,y) = \begin{cases} min(\{W(p) \mid p \text{ is a path from } x \text{ to } y\}) & \text{if there is a path from } x \text{ to } y \\ \infty & \text{otherwise} \end{cases}$$

A *shortest path* from x to y is any path p from x to y with weight $W(p) = \delta(x,y)$ and $W(p)$ is the *length* of the shortest path.

The *single-source shortest path problem*, which we call just the *shortest path problem* in the context of this thesis, is the following: given a graph $G = (V, E)$ and a source vertex $x \in V$, give a shortest path from x to each vertex $y \in V^*$, where $V^* \subset V$.

Example 2.1. Figure 2.1 shows an undirected graph. With the given edge weights the shortest path from e.g. A to I is $P_1 = \{A, C, D, F, G, H, I\}$ with length 2 and from B to F the path is $P_2 = \{B, A, C, D, F\}$ with length 1. Figure 2.2 shows a directed acyclic graph. Here, the path P_1 is still valid, but P_2 is not since the graph does not contain the edge (B, A). The shortest path from B to F becomes $P_3 = \{B, F\}$ with length 2. The graph of 2.2 does not contain a cycle (which can be determined with a visual inspection by noting that there are no edges that would go from left to right in the graph). However, changing the direction of e.g. (C, D) to (D, C) and (D, F) to (F, D) would create a cycle in the graph: $P_4 = \{C, E, F, D, C\}$.

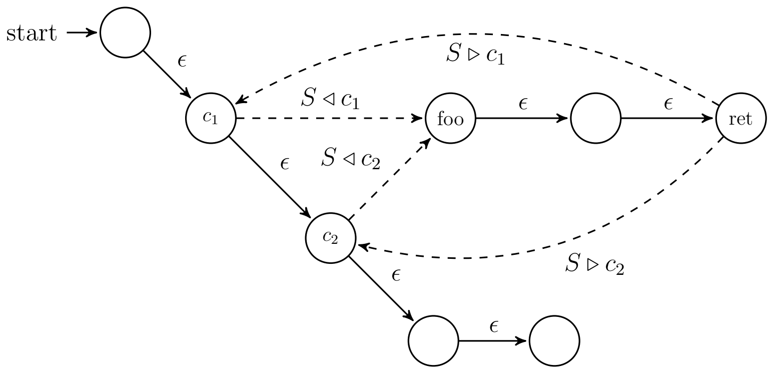

Figure 2.3: A pushdown automaton.

2.1.2 Pushdown Automata

Here we show how to model a call stack of a program with pushdown automata. The definitions follow that of [55].

Pushdown automata (PDA) constitute one of the three well-known *computational models*. The automata are like nondeterministic finite automata but have an extra component called a stack. A pushdown automaton can read and write arbitrarily many symbols on the stack with the limitation that the reading and writing can only be done on the top of the stack, in a “last-in-first-out” fashion. Pushdown automata are strictly more powerful (as a computational model) than *finite automata* (that do not have a stack) and strictly less powerful than *Turing machines* (which allow writing to an arbitrary location on a tape).

Formally, a pushdown automata is a 6-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$, where Q is the set of states, Σ is the input alphabet, Γ is the stack alphabet, $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function, $q_0 \in Q$ is the start state, and $F \subset Q$ is the set of accepting states. Q, Σ, Γ are finite sets and the initial stack is empty.

Since we are particularly interested in modeling a call stack structure we skim through a lot of details here and reduce our pushdown automata in the following way: we set Q as the states of our program, $\Sigma = \epsilon$, Γ as strings of the form $(s_1, s_2, ...)$, where $\forall i : s_i$ corresponds to a procedure call of the program, δ to correspond to transitions in the program, q_0 to correspond to the initial state of the program and $F = \emptyset$.

Example 2.2. Figure 2.3 shows a pushdown automaton (reflecting a control flow structure of a program) in a graph-form. Here, transitions with the label “ϵ” denote transitions which do not affect stack contents, “$S \triangleright x$” denote reading (and removing) the top element x of the stack and “$S \triangleleft x$” denote writing the element x to the top of the stack. We have two states - c_1 and c_2 - in the program corresponding to the call of the procedure “foo”. The call is modeled by writing suitable information of the caller to the stack. The stack plays an important role at the end of the procedure call - the “ret” node - where only the transition which reads the correct caller information is allowed. With finite automata such information would not be available, allowing us to return to a call point different to the one where the procedure was called from.

2.1.3 Fixpoints

Here we give a definition for the least fixpoint and a method for finding it. The definitions follow that of [44].

Definition 2.5 (Posets). A *partial order* over a set L is a relation $\sqsubseteq \subset L \times L$ that satisfies the following properties ($\forall l, l_1, l_2, l_3 \in L$): $l \sqsubseteq l$ (reflexivity), $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$ (anti-symmetry), and $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$ (transitivity).

If $\sqsubseteq$ is a partial order over L then $(L, \sqsubseteq)$ is a *partially ordered set* or a *poset*.

A poset $(L, \sqsubseteq)$ is *complete* if each of its subsets has a supremum and it contains a least element $\perp$ i.e. element that is less or equal to any other element in L.

Having $(L, \sqsubseteq)$ as a poset and $Y \subset L$, then Y is a *chain* if $\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$ (total ordering). We can pick a sequence of elements from L: $(l_1, l_2, ...)$. The sequence is an *ascending chain* if $\forall n, m \in \mathbb{N} : n \leq m \Rightarrow l_n \sqsubseteq l_m$.

Definition 2.6 (Fixpoints). Having a function $f : X \to X$, $x \in X$, x is a *fixpoint* of f if and only if $f(x) = x$. The *least fixpoint* of f is a fixpoint that is equal or less than all the other fixpoints of f.

The existence of the least fixpoint is usually guaranteed by the well-known Knaster-Tarski theorem. However, the theorem works on complete lattices (complete posets with subset infimums and a greatest element) and is not constructive in the sense that the least fixpoint cannot be straight-forwardly constructed by following the proof. To actually find the least fixpoint we utilize another well-known theorem called the *Kleene fixed-point theorem*:

Definition 2.7 (Kleene fixed-point theorem). Let L be a complete poset, and let $f : L \to L$ be a continuous (monotonic) function. Then the least fixpoint of f is the supremum of the *ascending Kleene chain* of f:

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq ... \sqsubseteq f^{(n)}(\perp) \sqsubseteq ...$$

Example 2.3. The set of non-negative integers $S = \mathbb{Z}_+ \cup \{0\}$ together with the normal ordering $\leq$ form a poset $(S, \leq)$. The poset is complete with the supremum of each $s \subset S$ being the maximal element of s and $\perp = 0$. Picking sequences of elements from S, we can form an ascending chain $(0, 1, 2, 4, ...)$ or a non-ascending chain $(3, 1, 2, 5, 0, ...)$.

For $f(x) = (x-2)^2$ (f is not monotone), we have two fixpoints, $f(x_1) = x_1 = 1$ and $f(x_2) = x_2 = 4$, and x_1 is the least fixpoint of f (but x_2 is not).

For $g(x) = x + \lceil |10 - x|/2 \rceil$ (g is monotone), we can find the least fixpoint with the ascending Kleene chain: $\perp = 0 \leq g(0) = 5 \leq g(5) = 8 \leq g(8) = 9 \leq g(9) = 10 \leq g(10) = 10 \leq ...$, giving 10 as the least fixpoint.

2.2 Program Analysis Methods

Here we discuss methods for program analysis. Two methods are prominent in the thesis: symbolic execution and flow analysis.

2.2.1 Symbolic Execution

Background

As stated in the introduction, *symbolic execution* is a program analysis technique that allows execution of programs using symbolic input values, instead of actual data. Then, during execution, the values of program variables have *symbolic expressions* that can represent several actual data values at once. [49]

This makes symbolic execution an attractive option to handle the effects of *state explosion* [59, 16]: if analysis is done with actual data values the analysis system must execute through paths corresponding to all possible enumerations of input values. The analysis basically branches for each possible choice of data value. Symbolic execution reduces the amount of paths to execute so that branching is only needed on branching structures of the program (such as if-then-else constructs).

The birth of symbolic execution is attributed to four designs from the 1970s: SELECT [11] which analyzes programs written in a subset of LISP, EFFIGY [38] which analyzes programs in PL/I-like languages, a system that analyzes programs written in FORTRAN [19] and DISSECT [35] which also analyzes programs in PL/I-like languages. Recent tools in the field include DART [30], CUTE [52], EXE [14] and its successor KLEE [13], all of which analyze programs written in C.

There seems to be no particular standard definition of a symbolic execution system. As shown here, symbolic execution systems are usually presented as systems for a particular programming language.

The systems do share some common elements: the usage of symbolic expressions as data values and the usage of a decision procedure of some sort to map symbolic expressions back to actual values. The systems differ in their selection of the language that the analysis is based on, in the way state space search is organized and in decision procedure expressivity. For an example, DART employs a linear constraint solver whereas EXE employs a system that converts a rich set of constraints to propositional satisfiability problems. A recent survey discusses the common elements and current development issues of symbolic execution. [49]

Another way to look at symbolic execution is to view it as a part of the field of *model checking*. This depends on how the term “model checking” is interpreted. In some interpretations it refers to techniques for solving the *model checking problem*: given a structure (with an initial state) and a property, decide if the property holds in the structure [17]. In other interpretations it refers to a wide collection of automatic method for finding errors in software. In any case, when symbolic execution is applied, the analysis problem it tries to solve can have relations to the model checking problem. For a gentle introduction into the many contemporary issues in the field see [60].

Test Generation

We are particularly interested in the way symbolic execution can be used to drive test generation [40]. As a high-level description we focus on a solution similar to EXE [14].

The essential idea is to store information about the actual values the symbolic variables can represent with *path conditions*. These are collected when symbolically executing program fragments that can affect the values with the idea that a symbolic execution path is a valid execution path in the actual program if and only if there is a solution (mapping symbolic values to actual values) to the set of path conditions generated for that path. Finding the solution to the constraint sets requires a decision procedure, such as the STP solver that EXE uses [14].

In addition to deciding which execution paths are valid the solution to the path condition set can be used to generate tests: the execution path gives a sequence of events and the solution gives data values to events. In particular, for testing purposes where we want to reach into certain target points of the program the symbolic execution can find execution paths from the starting point of the program to the target points and solve the path condition sets for data values.

We have described a symbolic execution system that does *forward analysis*. In contrast, a symbolic execution system that does *backward analysis* would start at the specified target points in the program and symbolically execute backwards trying to reach the starting point of the program and collecting path conditions in the same way as the forward analysis. We focus on just the forward analysis in the thesis.

Problems

Symbolic execution suffers from the following (as-of-yet) unsolved problems:

1. The time taken by the decision procedure to solve a set of constraints is generally exponential to the size of the program. There is some distinction between "easy" and "difficult" constraint problems based on whether they are solvable in polynomial time or whether they are NP-complete, e.g. linear programming being polynomial but integer programming NP-complete [46]. Furthermore, in certain cases constraint solving becomes undecidable [6]. To alleviate the cost of constraint solving many solvers use a timeout mechanism so that when certain time has passed the system defaults to a certain answer. With path constraints, the system must declare the set as unsolvable on timeout. This can effectively cut a valid execution path from the analysis, and with such a limitation symbolic execution can only underapproximate the actual semantics of the program that it analyzes.
2. The amount of separate paths to analyze can be exponential to the size of the program, which is referred to as the *path explosion problem* [33]. According to [49], *path merging* - where separate path information is stored into disjunctive constraints - can be used for significant speedup, however [33] claims that the resulting increase in constraint solving time does not warrant the use of the technique.

3. When symbolic execution is applied to systems with infinite-length execution paths the analysis must be limited to make it terminate in finite time. Usually this is done by using a heuristic to limit the search to some bounded length. A simple heuristic is to set a fixed limit to path length, the method used largely in bounded model checking [10]. The limit is another source of underapproximative behavior.

Precision/Efficiency Tradeoff

Summarizing from the last section, symbolic execution is in general either exponential-time (or even impossible in the case of undecidable constraint problems) or underapproximative. This requires a *precision/efficiency tradeoff* in design.

2.2.2 Flow Analysis

Background

Flow analysis is a program analysis method that can be used to statically gather semantical information of programs. Traditionally, flow analysis has been used in compiler technology to allow numerous optimizations, such as live variable analysis and constant folding [3]. The common element to these optimizations is the idea that the compiler can avoid generating code for parts of the program's semantics that can be retrieved without executing the actual code.

Flow analysis is a general term for various static analysis methods. We consider two variants: control flow analysis and data flow analysis. In *control flow analysis* (CFA), we try to determine what "elementary blocks" of the program can lead to what other "elementary blocks". In *data flow analysis*, we try to determine the possible data values that go in and out of "elementary blocks". Analysis methods vary in the program properties that they can handle. *Intraprocedural analysis* handles the semantics of a single procedure of a program whereas *interprocedural analysis* handles the interconnected behavior of multiple procedures. Some methods handle only single-threaded programs, some handle also multithreaded ones. Methods can be sensitive or insensitive to a few properties. For *flow-sensitive* methods the order of the statements of the program matters. For *context-sensitive* methods the calling context of a procedure matters. k-CFA denotes *k-order control flow analysis* i.e. control flow analysis with k levels of calling context. *Path-sensitive* methods note the possible difference in data values when doing a selection of branches. [44]

Flow analysis has also found its use in model checking in the form of tools that check temporal safety properties i.e. properties for which the counterexamples are finite execution traces (which themselves are results for reachability properties). The tools are directed to analyzing programs written in the C language and include SLAM [8], BLAST [9] (which also does test case generation), ESP [24] and a system built on top of ESP that uses a form of symbolic execution to disprove parts of the result of flow analysis [32]. Also, ZING [5] is able to analyze concurrent multithreaded programs.

Abstract Interpretation

Constructing a scalable flow analysis implementation requires special care, particularly when analysis is to be done on infinite-state systems. There is a commonly used framework for the task called *abstract interpretation* [22].

Abstract interpretation is a study of *abstractions*. The generic idea behind abstractions is to *soundly* approximate complex structures with simpler ones [44] (unsound abstractions also exist [4]). For program analysis this can correspond to *domain abstraction*, where the domain of a variable can be changed to a simpler one, when preserving the all the data values of the variable but possibly adding some values that the variable did not have in the original domain. This adds to the semantics of the program under analysis, therefore making the analysis method overapproximative.

The symbolic execution technique of representing multiple actual values with a single symbolic value can be seen as a form of abstraction. A survey suggests abstract interpretation as a way of scaling symbolic execution [49].

The de facto standard of doing flow analysis with abstract interpretation is to construct data domains with ordered sets such as lattices, to specify operations of the program with monotone functions over these sets and to calculate the least fixpoint of the functions to get the semantics of the program [27]. Recalling the definition of the least fixpoint from Section 2.1.3, the calculation is guaranteed to terminate when the domains satisfy the ascending chain condition:

Definition 2.8 (Ascending Chain Condition). A poset $(L, \sqsubseteq)$ satisfies the *ascending chain condition* if every ascending chain $(l_1, l_2, ...)$, where $\forall i : l_i \in L$, is eventually stationary, i.e. $\exists n \in \mathbb{N}, n > 0 : \forall m \in \mathbb{N}, m > n : l_m = l_n$.

For domains that do not satisfy the condition, convergence can be accelerated with a technique called widening [21]:

Definition 2.9 (Widening). A *pair-widening operator* or just *widening operator* for a poset $(L, \sqsubseteq)$ is a binary operator $\nabla : L \times L \to L$ that satisfies the following properties: $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_1 \nabla l_2 \wedge l_2 \sqsubseteq l_1 \nabla l_2$ (covering) and for every ascending chain $(l_1, l_2, ...)$ the ascending chain defined by $m_1 = l_1, m_{i+1} = m_i \nabla l_{i+1}$ stabilizes after a finite number of terms (termination).

Producing a Control Flow Graph

The previous section discussed how program semantics can be calculated. Usage of a control flow graph (CFG) is a common way to collect the semantics. We now discuss a few reference systems that create a control flow graph through flow analysis.

The control flow reconstruction of a low-level language is discussed in [37] with a particular notion that the results of control flow analysis and data flow analysis are intertwined: the control flow of the programs affects the data flow and vice versa.

This notion is even more apparent in [53], where the control flow analysis of programs written in Scheme is discussed. Here, the problem is that to do flow analysis of indirect lambda procedure calls precisely (the purpose of which is to compute a control flow graph) a control flow graph would already be needed. The problem is solved with overapproximation. A later publication [54] from the same author refines the implementation and marks two instantiations of it as 0-CFA and 1-CFA.

Generalizing to all higher-order languages, [36] shows that in general, a k-CFA is exponential-time when $k > 0$. By utilizing overapproximation they also show how to construct *polynomial k-CFA* systems with 0-CFA in $O(n^3)$ and 1-CFA in $O(n^6)$.

Another system for analyzing programs written in Scheme gives also a sub0-CFA analysis in linear time [7].

Multithreaded programs are handled in [50] and also *procedure summarization*, a technique where the results obtained by an analysis of a procedure call can be reused when analyzing another call to the same procedure, is discussed.

As an alternative approach to using explicit control flow graphs, a graph-free approach [42] computes a maximal fixed point solution to data-flow analysis without using a graph and claims less burden on memory.

Problems

As seen in the last chapter, k-CFA is in general exponential-time when $k > 0$.

Flow analysis also suffers from a few properties common to all static analysis methods:

1. Static analysis is undecidable in general [39], and
2. For multithreaded programs, static analysis is undecidable if it is both context-sensitive and synchronization-sensitive (i.e. precise w.r.t. the synchronization of the threads) [51].

Precision/Efficiency Tradeoff

Again, summarizing from the last sections, flow analysis is in general exponential-time (or even impossible in case of certain structural features) or overapproximative. This requires another *precision/efficiency tradeoff* in design. In this sense, flow analysis can be seen as a dual of symbolic execution.

Chapter 3

Model-Based Test Generation

This chapter describes how model-based test generation is done in Conformiq Qtronic. Specifically, we look at what kind of models the software takes as an input and give a graph-based representation for the explorable state space of the models. Then, we look at how tests can be generated from the models by running symbolic execution and further refine how the search space is kept finite by using a heuristic to bound it.

3.1 Test Generation in Conformiq Qtronic

We touched on Conformiq Software's model-based test generation product Qtronic [1] in the introduction. Now we look at the tool in more detail and describe where the tool might differ from other model-based testing solutions.

The tool's development was started circa 2004 and it has been in production use since 2007. During this time a lot of effort has been invested in the test generation algorithms of the tool. Because of this we can say with some confidence that the core test generation capabilities of the tool do not contain the kind of trivial implementational inefficiencies that can be associated with new systems. Still, the performance of the tool is an important factor of the tool's success and better algorithms for test generation are sought for.

The tool generates tests by finding concrete execution paths from the initial state of the model to syntactic control flow points of the model. The syntactic control flow points can be viewed e.g. as lines of the code of the model, however the tool allows the user to customize which points of the model become targets for the search. The concrete execution paths are given by a symbolic execution system that is described in the following sections. Test cases are generated by taking these paths and the data values related to them and converting the output to a suitable form.

3.2 Models

As stated in the introduction, a *model* is a mathematically structured form of functional requirements for a piece of software. A natural way to give these requirements is in a program form, and philosophically speaking, the model is a "golden design" of the actual system - a design to which all other designs are compared to - and is taken to be fault-free. In reality models are made manually and can contain error of thoughts just like actual systems. However, the gain from using models is that the inner workings of the system that are not relevant to testing can be abstracted away.

From a practical point of view, this view of the model essentially makes it just a program like any other. There is one difference: in our black-box testing approach we only have access to the external interface of the system-under-test, which is taken here to correspond to its inputs and outputs. The other side of the interface can correspond to, for an example, a user pressing keys and reading output from a monitor or another program that communicates with the system-under-test, but is left unspecified. To this effect the models contain abstracted input and output operations.

The models given to Qtronic are written in a combination of Java [31] and UML [45]. Before running analysis algorithms on a model the constructs are translated into CQλ, an in-house proprietary language that is effectively a subset of Scheme [2] with added capabilities such as the abstract read and write operations discussed earlier. The point of the translation is to reduce the amount of differing operations in the input language allowing analysis systems to be written with less code. The models given to the tool model complex real-world systems and can have features such as recursion and multithreaded behavior.

As a model is executed, we come upon abstracted input and output operations. An abstracted input operation of a model is given special consideration since it corresponds to a situation where the system-under-test needs an input from some external instance before execution can proceed.

Definition 3.1 (Stable states). A state of execution of a model is a *stable state* if and only if execution cannot proceed from the state without applying external input through an abstracted input operation.

In single-threaded models all abstracted input operations correspond to stable states, whereas in multi-threaded models all threads have to have run into abstracted input operations for a stable state to occur.

Because of the translation from Java and UML to CQλ the syntactic control flow points of the original model are mapped into the model given in CQλ. The targets of the search (the set of syntactic control flow points of the original model) are mapped into operations called checkpoints in the CQλ model.

Definition 3.2 (Checkpoints). A state of execution of a model is on a *checkpoint state* if and only if the operation that would be executed next is a *checkpoint operation*, an operation that corresponds to arriving on a marked syntactic control flow point in the original model. We call both checkpoint states and checkpoint operations *checkpoints* with the idea that a checkpoint points a place in model code that is a target for the symbolic execution search.

```
 1: ...
 2: public foo(x)
 3: {
 4:   int n=read();
 5:   if (n>0)
 6:   {
 7:     n=n+1;
 8:   }
 9:   else
10:   {
11:     n=n*-1;
12:   }
13:   for(int i=0;i<n;i++)
14:   {
15:     x=x*read();
16:   }
17: }
18: ...
```

Figure 3.1: Excerpt from an example model in Java.

```
 1: ...
 2: (define foo
 3:  (lambda (x)
 4:   (begin
 5:    (checkpoint proc:foo)
 6:    (let ((n (read)))
 7:     (begin
 8:      (if (> n 0)
 9:       (begin
10:        (checkpoint b:id=1,true)
11:        (set! n (+ n 1)))
12:       (begin
13:        (checkpoint b:id=1,false)
14:        (set! n (* n -1))))
15:      (letrec ((loop (lambda (i)
16:        (if (< i n)
17:         (begin
18:          (set! x (* x (read)))
19:          (loop (+ i 1)))))))
20:       (loop 0)))))))
21: ...
```

Figure 3.2: Translation of the model in Figure 3.1 into CQλ.

Example 3.1. Figure 3.1 shows an example model excerpt (a single procedure) in Java and Figure 3.2 shows its translation to a CQλ-like language. The `read()` calls in the Java model on lines 4 and 15 are translated into `(read)` calls in the CQλ model on lines 6 and 18, correspondingly. These calls read from an external input and, assuming that the model contains just a single thread, we can say that the execution of a model is on a stable state when these calls are to be executed next.

The resulting CQλ model also contains checkpoints. Here, the user of the tool has decided to have the method call on line 2 and the branching point on line 5 of the Java model as a syntactic control point worth of interest. This has resulted in added checkpoint operations on line 5 for the procedure call and on lines 10 and 13 for the branching point in the CQλ model.

We can also see that the for-loop on line 13 of the Java model has been translated into a (tail-)recursive procedure call in the CQλ model. Iteration is handled through recursion in CQλ.

3.3 State Space Representation

We can view the state space of a CQλ model abstractly by considering the states and transitions of the program the model represents. A state in the program consists of a program counter giving the location where execution is in the program code and the memory contents obtained by executing the program up to that point. Transitions correspond to moving from a state to another by a single step of execution.

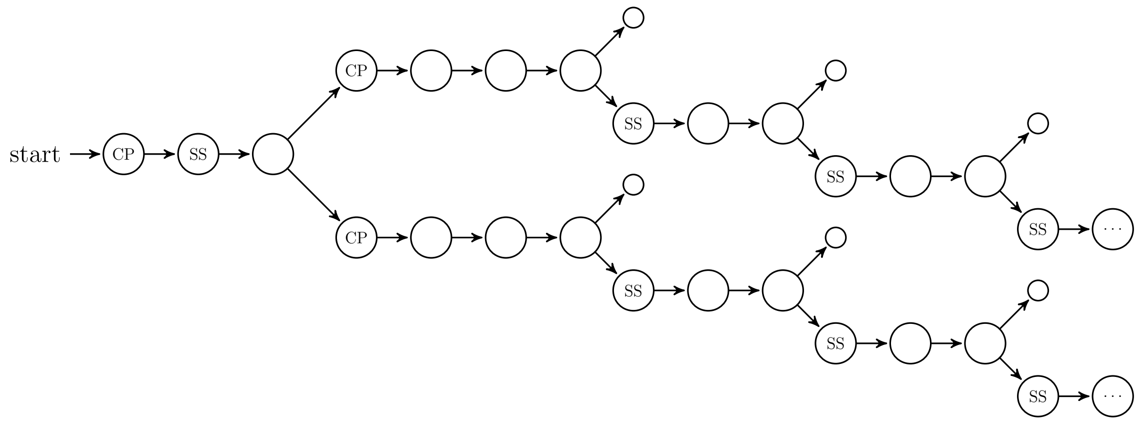

Figure 3.3: Abstract state space of the program in Figure 3.2.

The state space of the program can be represented as a graph. Each possible state is represented by a node in the graph and transitions between states are represented by a directed edge from the source state to the destination state in the graph. The graph becomes infinite when the state space of the program is infinite.

We mark the nodes in the graph with a label based on the type of the operation the program counter points to at the state which the node represents. We recognize the following types: stable states (with a label "SS"), checkpoints (with a label "CP") and interim operations (with an empty label). Interim operations contain all operations that do not correspond to stable states or checkpoints.

Example 3.2. Figure 3.3 shows the abstract state space representation of the program excerpt given in Figure 3.2. Starting from the beginning of the procedure "foo", the execution runs first into the checkpoint for the procedure call on line 5, which is reflected in the graph by a checkpoint node. After that, on line 6 there is a call to read(), which is reflected in the graph by a stable state node. Then, the if-then-else construct starting on line 8 causes a branch in the state space and after that, on lines 10 and 13 there is a checkpoint for each branch. The execution then proceeds to the recursive loop call. Since the terminating condition for the loop is based on comparing a counter to the value of an abstract input the counter is expected to run to an arbitrary number thus requiring an arbitrary number of iterations through the loop. This makes the state space infinite and, reflecting this, the graph contains infinite amount of structures comprised of the comparison, the stable state and the counter increment in the loop.

3.4 Symbolic Execution

Implementation

The test generation system in Qtronic is based on symbolic execution. The implementation follows much of the theoretical basis given in Section 2.2.1 - sharing also the problems of symbolic execution mentioned there.

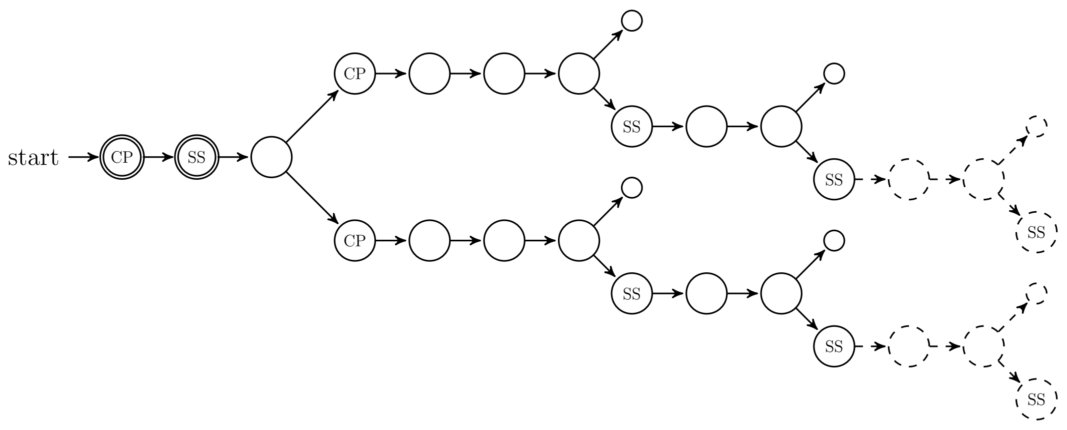

Figure 3.4: Symbolic execution state space search of the program in Figure 3.2 with *Depth* = 1 (double circle), *Depth* = 2 (single circle) and *Depth* = 3 (dashed).

We do not have the need to identify exactly how the state space search is organized in the implementation. This allows us to abstract out details such as the timeout mechanism used in the decision procedure. We will, however, mention that the implementation does not do path merging since this affects the amount of branches in the state space and therefore the amount of paths the symbolic executor needs to execute through.

The detail that plays a particular importance in the thesis is the way in which the execution is bounded. The bound is necessary since (as shown in the last section) the models that are analyzed can have paths of infinite length.

Definition 3.3 (Bound). Qtronic employs the following heuristic to bound search: we have an integer-valued counter k that is initially set to an analysis-wide fixed positive integer bound *Depth*. The counter is duplicated for each symbolic execution path. When executing a path if the execution comes to a stable state the counter for that path is reduced by one and if the result is zero the path is not executed further. If the execution comes to a checkpoint that has not yet been covered, a decision procedure is called to check that the path is a valid actual execution path of the model, the checkpoint is added to the set of covered checkpoints and the counter is reseted to *Depth*.

Example 3.3. Figure 3.4 shows how the bound affects the symbolic execution state space search on the program of Figure 3.2. Here we have adopted the abstracted state space representation format that was discussed in the last section and shown in Figure 3.3. Here, the part of the graph drawn with double circles corresponds to the state space symbolic execution explores with $Depth = 1$. Initially, $k = 1$. We run first into a checkpoint which resets $k = 1$. Then we run into a stable state which decreases $k = 0$ and the execution is stopped. The part of the graph drawn with single circles corresponds to exploration with $Depth = 2$. Here, $k = 1$ after the first stable state and the execution runs into a checkpoint on both branches of the if-then-else construct resetting $k = 2$. The execution can then proceed through the iteration loop two times before running into a second consecutive stable state where $k = 0$. With $Depth = 3$, the execution can reach one more iteration loop.

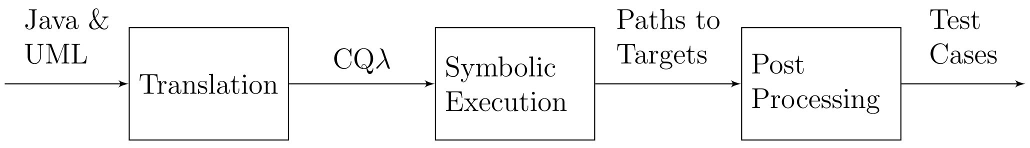

Figure 3.5: The test generation process in Qtronic.

Choosing the heuristic to depend on the stable states and checkpoints on a path has a difference to a bound which would be decreased by each step of the execution. Here, the paths can be arbitrarily long as long as they do not contain stable states.

Figure 3.5 summarizes the test generation process in Qtronic.

Chapter 4

Cutting Symbolic Execution Search

In this chapter we give the main contribution of this thesis, a system for cutting branches of the symbolic execution search based on flow analysis information.

4.1 Flow Analysis of a Model

We consider flow analysis first. As seen in the introduction, an analysis method that searches through execution paths from a program's initial state to some targets (i.e. forward analysis) cannot predict which paths turn out to be relevant for the search. The motivation for using flow analysis (or any static analysis method) to help predict path relevance is that by safely overapproximating the program's semantics we get information that can be used to disprove properties of the program, e.g. disprove path relevance.

4.1.1 Control Flow Graph as a PDA

We now discuss our implementation of a flow analysis system.

As discussed in Chapter 2, symbolic execution and flow analysis both need to do an precision/efficiency tradeoff in design. To select the correct precision level for these systems we need to take into account our task of test generation. Because symbolic execution is used to generate tests with hopefully good coverage, the goal should be in making symbolic execution more and more precise over time. Flow analysis should also be as precise as possible but using it to optimize symbolic execution gives the natural constraint that it must be more efficiently computable than symbolic execution. Considering the exponential-time nature of precise symbolic execution we decide to demand that a viable flow analysis system must be computable in polynomial-time.

As stated in the last chapter, the input programs for the symbolic execution system are given in CQλ, a subset of Scheme with added operations for symbolic inputs and outputs. We use CQλ also as the input language for flow analysis.

The actual implementation has been a joint work of several individuals at Conformiq Software, including the author. Due to size constraints, we rely on literature on many of the implementation details. The implementation is mostly based on the control flow analysis implementation for Scheme given in [7], in particular the collecting machine and its abstractions-using version named abstract machine in the paper. The implementation collects the semantics of a program by calculating a least fixpoint in the manner given in section 2.2.2. The implementation is a polynomial 1CFA.

We chose to output the control flow graph given by the flow analysis algorithm as a simplified graph-form pushdown automaton of Section 2.1.2. The reason for choosing PDA over finite state machines and Turing machines is that since we are particularly interested in execution paths finite state machines would not suffice in giving precise enough information of paths and that producing a Turing machine equivalent structure would not simplify the problem over the original structure given in CQλ.

The PDA graph consists of a directed acyclic graph for procedures of the CQλ program and the connections between the DAGs. In 1CFA, procedures are considered as separate if they correspond to different lambda procedures or if their call contexts differ in the first order i.e. if they have been called from different call points. Since the program can contain a finite amount of lambda procedures and call points, this makes the amount of separate procedures finite. To reflect this, the graph contains a DAG just for each separate procedure.

The flow analysis implementation combats the path explosion problem discussed in 2.2.1 by merging branches at the end of their scope. This causes imprecision but is a viable approach since flow analysis does not need to collect constraints for the paths.

The PDA graph has labels for nodes corresponding to stable states, checkpoints and call points. Other details of the states are abstracted out.

Example 4.1. Figure 4.1 shows the resulting PDA control flow graph after 0CFA flow analysis of the program in Figure 3.2. Since 0CFA does not take into account call contexts at all, we have just two separate procedures: the one corresponding to the procedure “foo” and the one corresponding to the procedure “loop”. The branches coming from the if-then-else construct on line 8 of the program are merged at the end of the scope of the construct on line 14.

4.1.2 Analysis

We have obtained a polynomial-time algorithm that computes a PDA control flow graph that is a safe overapproximation of the reachable state space of a Scheme-like program.

Since the approximation is sound, it holds that for each valid actual execution path of the program there is a corresponding path in the control flow graph. For target checkpoints, it follows that if there is a valid actual execution path of the program that reaches a checkpoint, there must be a path in the control flow graph that includes

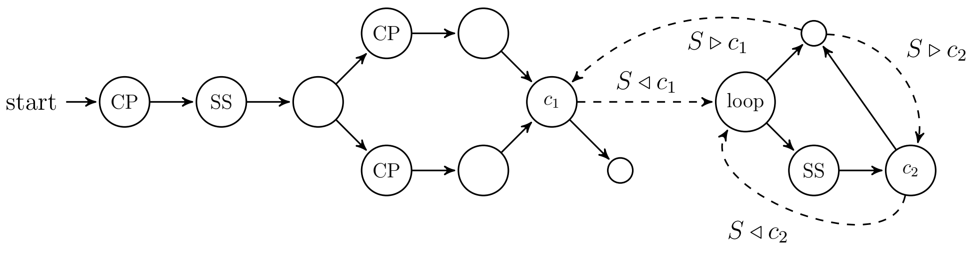

Figure 4.1: Control Flow Graph of the program in Figure 3.2 as computed by 0CFA flow analysis.

the node for that checkpoint. Conversely, if a target checkpoint cannot be found in the control flow graph, it cannot be reached with any valid actual execution path.

This allows us to make an optimization to the symbolic execution state space search: if the intersection of the set of yet-to-be-covered target checkpoints and the set of checkpoints in the control flow graph becomes empty, the search can be stopped. This optimization is very similar to dead code elimination of compiler technology [3].

This optimization could in itself help to make symbolic execution faster, but we can do more by utilizing the information given by the bounding heuristic of the symbolic execution system. In addition to cutting search when no checkpoints can be reached anymore we can cut search on an individual branch if we can determine that continuing execution on the branch cannot reach new checkpoints within the given bound. We calculate shortest paths in the control flow graph to do such a determination.

4.2 Cutting Search

We assume here that by utilizing just the control flow graph, not much can be said about whether individual branches under execution by the symbolic execution system are going to reach new checkpoints. We therefore require the availability of some dynamic information of the symbolic execution system, namely the "current execution state", the current set of yet-to-be-covered checkpoints and bounding heuristic information. The symbolic execution already needs to keep track of these items so this does not place an extra burden on the system.

4.2.1 Establishing Correspondence To Symbolic Execution

We are not assuming anything of the way the states of the search are represented in the symbolic execution system. Instead, we establish the correspondence of a "current execution state" of the symbolic execution system to nodes in the control flow graph via the usage of branch signatures (which we refer to as *correspondence tracking*).

Definition 4.1 (Branch Signatures). A *branch signature* is a sequence obtained by recording choices for branching events in an execution path.

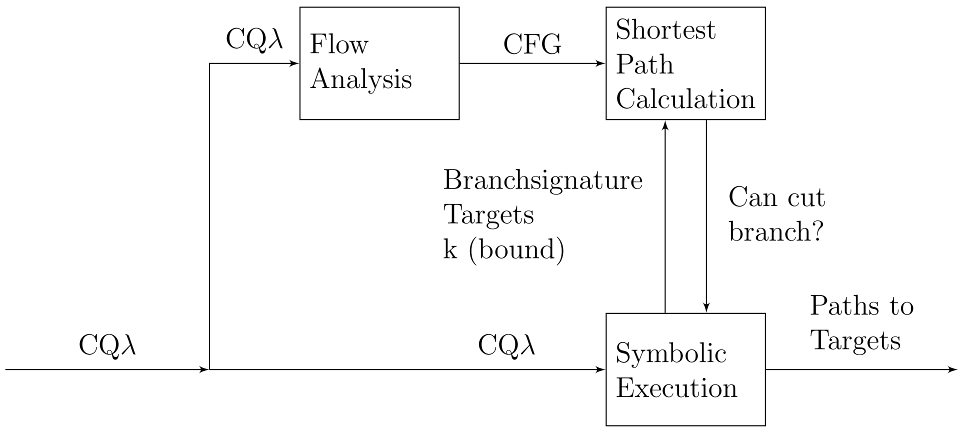

Figure 4.2: The branch cutting system.

Definition 4.2 (Tracking Points). A *tracking point* is a pair consisting of a node in the control flow graph and a stack containing procedure call nodes, corresponding to the call stack gathered during execution.

Example 4.2. Considering the program of Figure 3.2, the symbolic execution search could be in a state reached by executing a path where it has taken the then branch of the if-then-else construct on line 8, the then branch of the if-then-else construct on line 16 and then again the then branch of the construct on line 16. This would give a branch signature of the form (branch1:then, branch2:then, branch2:then), which can be used to track to the corresponding nodes in the control flow graph, arriving to a tracking point consisting of the stable state node of the "loop" procedure with stack contents of c_2 on top of c_1.

The usage of branch signatures restricts us to the successor nodes of branching points of the control flow graph. This is acceptable since we essentially want to cut branches of search at the starting point of the branch.

4.2.2 PDA Shortest Path Calculation

The shortest path calculation algorithm solves the following problem:

Given a control flow graph, a tracking point, a subset of nodes of the graph corresponding to the current target checkpoints and the current bound heuristic counter value k, is there a shortest path from the current node to a node in the target checkpoint nodes of length less or equal as k with edge weights given by the given heuristic?

We give an algorithm to the shortest path problem presented here in the next chapter. For multithreaded models, we could have a tracking point for each thread of the model. We calculate shortest paths then by simply calculating the shortest path for each tracking point and taking the minimum of the results.

We have arrived at augmenting the symbolic execution part of the test generation process with the design of Figure 4.2.

Chapter 5

PDA Shortest Path Calculation

In this chapter we give an algorithm to calculate (single-source) shortest paths in a graph that represents a pushdown automaton (PDA) equivalent structure. We tune our algorithm to the requirements of the branch cutting system of last chapter but (as will be shown) it can be used without modifications on general PDAs.

5.1 Design Criteria

We note firstly that the single-source shortest path problem is well-known in algorithmic theory for finite graphs and efficient algorithms, such as the *breadth-first-search* (BFS), the *depth-first-search* (DFS), the *Dijkstra's algorithm* and the *Bellman-Ford algorithm*, exist for it [20]. In our problem instances, edge weights cannot be negative and because of the control flow structure there are not many edges leading out of any node. Therefore, a simple approach such as a DFS can be a starting point for our algorithm.

The real difficulty in our problem instance is that the stack size of the PDA is not bounded from above, making it possible to have infinite paths (without cycles in the sense that we can have the same node twice in a path but with different stack contents) in our finitely encoded graph. The reachability problem - of which our shortest path problem is a special instance - is undecidable in general over infinite graphs [57].

The helping factor is that we have an upper bound to the lengths of paths that are acceptable. Also, without loss of generality, the shortest path to a checkpoint does not contain cycles since if it would we could form another shortest path by removing the cycles [20]. This is extendable to cycles arising from the usage of the stack: if a shortest path would contain the same node twice but at the second time with a stack that would have the same elements plus some additional element(s) we could as well form another shortest path by removing the path between the two nodes. Shortest paths that contain the same node twice but with a stack that has less elements is not a problem since stack size is bounded by below (zero).

Therefore, we always have a finite shortest path to a checkpoint if such a path exists. The problem is that there can be infinite paths that do not contain any checkpoints.

As a clue to how to work around this we note that paths in a graph for a single procedure are finite.

The key to an efficient shortest path algorithm is utilizing the *optimal substructure property*: a shortest path between two nodes contains other shortest paths within it. To this end, usage of *greedy methods* (methods in which the locally best option is chosen) and *divide-and-conquer* (where the problem is divided into parts and solved recursively) or *dynamic programming* (where a subproblem of the problem is solved separately before proceeding with the original problem) techniques is suggested. [20, 56]

Structures that stay static between separate calls to the shortest path algorithm provide an attractive place to utilize the optimal substructure property. In particular, we know that the CFG and the bound heuristic stay constant during all executions of the algorithm, whereas the tracking point and the bound heuristic counter value generally change between each execution. Search targets change, but more sporadically.

5.2 Inputs

We summarize here the formalizations of the inputs of the algorithm. We denote the set of suitably encoded identifiers corresponding to the procedures of the model with `Procedures` and the set of suitably encoded checkpoint identifiers with `Checkpoints`.

The inputs are then given as a 5-tuple: $\langle$`CFG`, `TP`, `Targets`, w, $k\rangle$, where

- `CFG` is the control flow graph. We mark the whole graph with $G = (V, E)$ and graphs for individual procedures with $G_p = (V_p, E_p, p_{root})$, where $p \in$ `Procedures` and $p_{root} \in V_p$ is the root node of the procedure p. $\forall p \in$ `Procedures` : G_p is a directed acyclic graph.

  Procedures have as an attribute the set of procedures which call the procedure in question: $\forall p \in$ `Procedures` : $p.callers \subset$ `Procedures`.

  All nodes have as an attribute the type of the node: $v.type \in \{call, return, stablestate, checkpoint, other\}$.

  "call" nodes have as an attribute the identifier of the procedure they call: $v.procedureid \in$ `Procedures`.

  "checkpoint" nodes have as an attribute the identifier of the checkpoint: $v.checkpointid \in$ `Checkpoints`.

- `TP` is the current tracking point, i.e. the source node and stack contents from which the search is started. We mark it as `TP` $= (v, S)$, where $v \in V$ and S is a stack of elements of V. We denote pushing an element v to the stack S by $S \triangleleft v$ and popping the element by $S \triangleright v$.

- `Targets` is the set of current search target checkpoint identifiers: `Targets` $\subset$ `Checkpoints`.

- w is the bound heuristic, given as a *weight function* over the edges of the CFG: $w : E \rightarrow \mathbb{Z}_0$. In particular we have the heuristic of Conformiq Qtronic:
$$w((v_1, v_2)) = \begin{cases} 1 & v_1.\mathit{type} \text{ is stablestate} \\ 0 & \text{otherwise} \end{cases}$$
- k is the current bound heuristic counter value: $k \in \mathbb{Z}_0$.

The decision problem that the algorithm must solve is then: given $\langle$`CFG`, `TP` $= (v, S)$, `Targets`, w, $k\rangle$, is $\min(\{\delta(v, \hat{v}) \mid \hat{v} \in V \wedge \hat{v}.\text{type is checkpoint} \wedge \hat{v}.\text{checkpointid} \in$ `Targets`$\}) \leq k$?

Generalizing the algorithm to general PDAs is basically a terminology issue. Replacing procedure calls with transitions that push to stack, returns with transitions that pop from stack, checkpoints with general targets of search and the tracking point with the source state gives us the corresponding environment with basic PDA terminology. Generally, we need to drop the claim that graphs for single procedures are DAGs.

5.3 The Algorithm

We start by devising a method that allows us to "skip" procedure calls in search. The idea is that if the shortest path from the root node of a procedure to its returning nodes is known then a shortest path search routine that comes upon the procedure call can continue the search from the nodes succeeding the call without going through the part of the graph related to the procedure call. Of course, an exhaustive search must go through all accessible parts of the graph but the point is that skipping procedure calls allows us to e.g. limit the scope of a part of the search routine to a single procedure and therefore break the search problem into subproblems more effectively.

We use the dynamic programming method of *tabulation* to store shortest paths from root nodes of procedures to the corresponding return nodes. The immediate problem in determining such paths is that the paths can, and often will, depend on the shortest paths to return nodes in other procedures. To handle this we formulate the search as a fixpoint algorithm over single procedures in the following way: at first no paths are known and we have a table containing "undefined" for all the procedures. Then, we search for shortest paths on individual procedures. Paths are not allowed if they depend on an "undefined" value i.e. if the path would go through a procedure for which a shortest path has not yet been determined. Paths are accumulated to the table and the algorithm terminates when there are no more changes i.e. a (least) fixpoint has been reached. However, there is a slight twist. Instead of storing sets of paths to the table we use the minimal path length as the set's identifier.

This gives us the poset $((\bot \cup \mathbb{Z}_0), \sqsubseteq)$, with $\bot$ as the least element (corresponding to the "undefined" value) and $\sqsubseteq$ defined as: $\forall l_1, l_2 \in (\bot \cup \mathbb{Z}_0) : l_1 \sqsubseteq l_2$ iff $l_1 = \bot \vee (l_2 \neq \bot \wedge l_1 \geq l_2)$. It is easy to see that the poset is complete and that the order is in fact a total order. We also define $\forall n \in (\bot \cup \mathbb{Z}_0) : (\bot + n) = (n + \bot) = \bot$ to allow a bit shorter algorithmic representation.

A later realization gave us the idea that shortest paths from roots to checkpoints can be calculated in the same manner as was done with return nodes. This was added to the implementation to enable some optimizations that are discussed later.

Before presenting the algorithms we define a few helper structures and routines:

Worklist is a structure containing elements that are to be worked on. It provides operations for adding elements Worklist.put(elem) and taking elements out (and removing them from the list) Worklist.take(). We use instantiations of Worklist as containers that hold the search frontier.

Visited is a map of the form: $* \to \mathbb{Z}_0$, with instantiations of Visited fixing the parameter $*$. The idea behind Visited is to provide the search memory of what elements have been encountered, allowing the search to be cut on cycles. It is not mandatory for DAG structures such as the individual procedure graphs but we use it for generality (general PDAs with cyclic structures). Visited is modified through the procedure `UpdateVisited`, which returns "true" if the element has not been encountered before or if it is now encountered with a shorter path than before, and "false" otherwise.

```
Procedure UpdateVisited(elem, $n$)
  if $elem \notin$ Visited or not Visited(elem) $\leq n$ then
    Visited(elem) $\leftarrow n$
    return "true"
  end if
  return "false"
```

Table is the map: $(\texttt{Procedures} \times \{return, checkpoint\}) \to (\perp \cup \mathbb{Z}_0)$. Table contains shortest paths to return and checkpoint nodes on a per-procedure basis. Table is modified through the procedure `UpdateTable`.

```
Procedure UpdateTable($p$, type, $n$)
  if not $n \sqsubseteq$ Table($p$, type) then
    Table($p$, type) $\leftarrow n$
  end if
```

We define a partial order between tables as: $\text{Table}_1 \sqsubseteq \text{Table}_2$ iff
$\forall(p,\text{type}) \in (\texttt{Procedures} \times \{return, checkpoint\}) : \text{Table}_1(p,\text{type}) \sqsubseteq \text{Table}_2(p,\text{type})$ with the least element as a Table with $\perp$ in every field.

The method for calculating the table of shortest paths is given in Algorithm 5.1. Worklist is implemented here as a queue containing elements of `Procedures`. The implementation is a pretty straightforward realization of the abstract worklist algorithm given in [44]. On lines 11-14, the generic way to proceed would be to insert all procedures to the worklist. However, as an optimization we note that only the procedures for which the analysis can be influenced need to be inserted.

The single procedure shortest path analysis is given in Algorithm 5.2. Worklist is implemented here as a queue containing elements of $(V \times \mathbb{Z}_0)$, making the analysis a straightforward DFS.

```
Procedure ComputeTable(CFG, Targets, $w$)
Input: CFG, Targets, $w$
Output: Table (Table)
 1: Initialization: Worklist, Table are empty
 2: for all $p \in$ Procedures do
 3:    Worklist.put($p$)
 4:    Table($p$, $return$) $\leftarrow \bot$
 5:    Table($p$, $checkpoint$) $\leftarrow \bot$
 6: end for
 7: while Worklist not empty do
 8:    $p \leftarrow$ Worklist.take()
 9:    Newtable $\leftarrow$ AnalyzeProcedure(CFG, Table, $p$, Targets, $w$)
10:    if not Newtable $\sqsubseteq$ Table then
11:       Table $\leftarrow$ Newtable
12:       for all $caller \in p.callers$ do
13:          Worklist.put($caller$)
14:       end for
15:    end if
16: end while
17: return Table
```

Algorithm 5.1: Table calculation algorithm.

```
Procedure AnalyzeProcedure(CFG, Table, $p$, Targets, $w$)
Input: CFG, Table (Table), $p$ (Procedures), Targets, $w$
Output: Table (Table)
 1: Initialization: Worklist, Visited are empty
 2: Worklist.put($\langle p_{root}, 0\rangle$)
 3: while Worklist not empty do
 4:    $\langle v, n\rangle \leftarrow$ Worklist.take()
 5:    if $v.type$ = call then
 6:       UpdateTable($p$, checkpoint, $n$ + Table($v.procedureid, checkpoint$))
 7:       $n \leftarrow n$ + Table($v.procedureid, return$)
 8:    else if $v.type$ = return then
 9:       UpdateTable($p$, return, $n$)
10:    else if $v.type$ = checkpoint $\wedge$ $v.checkpointid \in$ Targets then
11:       UpdateTable($p$, checkpoint, $n$)
12:    end if
13:    for all $succ \in \{v_x \mid (v, v_x) \in E_p\}$ do
14:       if $n \neq \bot \wedge$ UpdateVisited($succ, n + w(\langle v, succ\rangle)$) then
15:          Worklist.put($\langle succ, n + w(\langle v, succ\rangle)\rangle$)
16:       end if
17:    end for
18: end while
19: return Table
```

Algorithm 5.2: Single procedure analysis.

With a method to skip procedure calls we can make use of an interesting observation: Suppose there is some shortest path between two nodes (with their accompanying stacks): (v_1, S_1) and (v_2, S_2). Then, starting from (v_1, S_1) and following the path we only run into places in which the stack is modified by pushing one element onto it or by popping one element out of it, making the stack "continuous" in some sense between the two nodes. More interestingly, there is some region in the path where the stack has the minimal amount of elements, denoted by e.g. $\hat{S}$, which must be the bottom part of both S_1 and S_2 (proof omitted).

Therefore, we can make a search routine that starts from (v_1, S_1) and skips procedure calls but proceeds normally otherwise, popping elements from the stack on returning from procedures. If the search is done in this way exhaustively we are guaranteed to arrive at some point to the point where the stack becomes $\hat{S}$. Of course, we do not know at which point this happens (as generally we do not know what the minimal stack is), and therefore we make the search routine's goal as collecting the set of nodes (with the accompanying path lengths) that lie on procedure calls (that we otherwise skip during search). If the routine finds the target during this, all the better. As mentioned in the design criteria section, the size of the stack is bounded from below (zero) and therefore the routine should terminate (if cycles not concerning the stack are taken care of).

Then we devise another search routine that does the opposite of the first one. We start with the set of nodes collected by the first routine and expand search from them by going into each procedure call, as well as covering the parts available when skipping the calls, but terminating search on procedure returns. We can do so since the only parts of the graph that we miss by expanding the search in this way are the places in which the stack would first have to popped and we know that one of the collected nodes must have the minimal stack of the shortest path in the beginning. As mentioned in the design criteria section, we can disallow cycles where the stack would grow uncontrollably, and since the graph is finite, the routine should terminate (if cycles not concerning the stack are taken care of).

Algorithm 5.3 gives the first full version of our shortest path calculation algorithm. Worklist is implemented here as a priority queue containing elements of $(V \times S \times \mathbb{Z}_0)$, with the path length as the key (allowing duplicate keys). We have also another instantiation of Worklist, Frontier, which is implemented as a priority queue containing elements of $(V \times \mathbb{Z}_0)$, with the path length as the key (allowing duplicate keys).

On lines 2-4, on the first run of the algorithm the `ComputeTable` procedure is run with the set of all checkpoints to initialize Table. Lines 5-23 correspond to the routine that skips procedure calls and adds the nodes corresponding to the calls to Frontier. Lines 25-40 correspond to the routine that terminates search on procedure returns. Tabularized shortest paths to checkpoints are used on line 27 where the search can be cut on procedure calls from which no paths to checkpoints exist.

```
Procedure SPCv1(⟨CFG, TP, Targets, $w$, $k$⟩)
Input: ⟨CFG, TP = $(v_{TP}, S_{TP})$, Targets, $w$, $k$⟩ (input formalism)
Output: "true" if found, otherwise "false"
 1: Initialization: Worklist, Frontier, Visited are empty
 2: if first run? then
 3:    Table ← ComputeTable(CFG, Checkpoints, $w$)
 4: end if
 5: Worklist.put($\langle v_{TP}, S_{TP}, 0\rangle$)
 6: while Worklist not empty do
 7:    $\langle v, S, n\rangle \leftarrow$ Worklist.take()
 8:    if $n \leq k$ then
 9:       if $v.type$ = checkpoint $\wedge$ $v.checkpointid \in$ Targets then
10:          return "true"
11:       else if $v.type$ = return then
12:          $S \triangleright v$
13:       else if $v.type$ = call then
14:          Frontier.put($\langle v, n\rangle$)
15:          $n \leftarrow n$ + Table($v.procedureid, return$)
16:       end if
17:       for all $succ \in \{v_x \mid (v, v_x) \in E\}$ do
18:          if $n \neq \perp \wedge$ UpdateVisited($\langle succ, S\rangle, n + w(\langle v, succ\rangle)$) then
19:             Worklist.put($\langle succ, S, n + w(\langle v, succ\rangle)\rangle$)
20:          end if
21:       end for
22:    end if
23: end while
24: clear Visited
25: while Frontier not empty do
26:    $\langle v, n\rangle \leftarrow$ Frontier.take()
27:    if $n \leq k \wedge v.type \neq$ return $\wedge$ not ($v.type$ = call $\wedge$
       Table($v.procedureid, checkpoint$) = $\perp$) then
28:       if $v.type$ = checkpoint $\wedge$ $v.checkpointid \in$ Targets then
29:          return "true"
30:       else if $v.type$ = call $\wedge$ UpdateVisited($v.procedureid_{root}, n$) then
31:          Frontier.put($\langle v.procedureid_{root}, n\rangle$)
32:          $n \leftarrow n$ + Table($v.procedureid, return$)
33:       end if
34:       for all $succ \in \{v_x \mid (v, v_x) \in E\}$ do
35:          if $n \neq \perp \wedge$ UpdateVisited($succ, n + w(\langle v, succ\rangle)$) then
36:             Frontier.put($\langle succ, n + w(\langle v, succ\rangle)\rangle$)
37:          end if
38:       end for
39:    end if
40: end while
41: return "false"
```

Algorithm 5.3: Shortest path calculation algorithm version 1.

```
Procedure SPCv2(⟨CFG, TP, Targets, $w$, $k$⟩)
Input: ⟨CFG, TP = $(v_{TP}, S_{TP})$, Targets, $w$, $k$⟩ (input formalism)
Output: "true" if found, otherwise "false"
1: Initialization: Worklist, Visited are empty
2: if first run? $\lor$ storedtargets $\neq$ Targets then
3:    Table $\leftarrow$ ComputeTable(CFG, Targets, $w$)
4:    storedtargets $\leftarrow$ Targets
5: end if
6: Worklist.put($\langle v_{TP}, S_{TP}, 0\rangle$)
7: while Worklist not empty do
8:    $\langle v, S, n\rangle \leftarrow$ Worklist.take()
9:    if $n \leq k$ then
10:      if $v.type$ = checkpoint $\land$ $v.checkpointid \in$ Targets then
11:         return "true"
12:      else if $v.type$ = return then
13:         $S \triangleright v$
14:      else if $v.type$ = call then
15:         if Table($v.procedureid, checkpoint$) $\neq \bot$ $\land$
            n + Table($v.procedureid, checkpoint$) $\leq k$ then
16:            return "true"
17:         end if
18:         $n \leftarrow n$ + Table($v.procedureid, return$)
19:      end if
20:      for all $succ \in \{v_x \mid (v, v_x) \in E\}$ do
21:         if $n \neq \bot \land$ UpdateVisited($\langle succ, S\rangle, n + w(\langle v, succ\rangle)$) then
22:            Worklist.put($\langle succ, S, n + w(\langle v, succ\rangle)\rangle$)
23:         end if
24:      end for
25:   end if
26: end while
27: return "false"
```
Algorithm 5.4: Shortest path calculation algorithm version 2.

The first version of the algorithm suffers from cases where no checkpoint can be found within the bound (but the search space is large). The optimization of line 27 is only able to cut search for procedures from which there is no path to any checkpoint. Recalling our use scenario, the set of target checkpoints is likely to reduce over time as paths to checkpoints are found, but this is not reflected in the search since the table of shortest paths was computed only in the beginning and can later on contain paths for checkpoints that are already found.

With this idea in mind, we devise another version of the algorithm in which the table of shortest paths is reseted when the set of target checkpoints change. This is a tradeoff and we must test whether the potential improvements in search times beat the overhead coming from executing `ComputeTable` more.

The second version of the algorithm is given in Algorithm 5.4. Worklist is implemented here in the same way as in version 1. The difference to the first version is the aforementioned resetting of the table of shortest paths on lines 2-4 and that on line 15 we do not add nodes to Frontier since we can immediately decide based on the table if there is a target checkpoint within the bound. Since we are not adding anything to Frontier we can forgo the second part of the first algorithm altogether.

5.4 Analysis

Full proofs for the termination and correctness of the algorithms are out of scope of the thesis. However, we try to give convincing arguments about some of the basic properties of the algorithms. Preliminaries:

- We base our analysis as complexity w.r.t. the size of the (node set of the) input control flow graph $G = (V, E)$, $|V| = n$.
- We assume that the amount of successors for nodes is bounded by a constant.
- $|$`Procedures`$|$ is linear to n.
- The maximal possible path that does not use the stack goes through each node of the graph. The weights of the edges given by the bound heuristic are bounded from above by some constant c. The maximal possible path length therefore becomes $c \cdot (n - 1)$, i.e. linear to n.
- When Visited is instantiated with the parameter being the set of nodes of the graph `UpdateVisited` can return "true" maximal path length amount of times for each node (for each node, the path length can first be set to the maximal path length and then reduced one by one), i.e. a quadratic amount of times to n. It is also easy to see that the runtime of `UpdateVisited` is in $\mathrm{O}(n)$. When Visited is instantiated with the parameter being the set of nodes and the associated call stacks the runtime of `UpdateVisited` becomes $\mathrm{O}(|S| \cdot n)$ and the amount of "true" returns $\mathrm{O}(|S| \cdot n^2)$.
- The comparison of tables in `UpdateTable` can be implemented as comparisons over sets of (`Procedures` $\times \{return, checkpoint\}$), and so the runtime of `UpdateTable` is in $\mathrm{O}(n)$.
- Table can be updated through `UpdateTable` maximal path length amount of times for each element of (`Procedures` $\times \{return, checkpoint\}$) (again, since the path length can first be set to the maximal path length and then reduced one by one), i.e. a quadratic amount of times to n.

`AnalyzeProcedure`:

- The `AnalyzeProcedure` algorithm computes a straightforward DFS to tabulate into Table the shortest paths to return and checkpoint nodes on a graph for an individual procedure of the model. The return value of `UpdateVisited` dictates which elements can be put on Worklist (lines 14-15) and therefore the loop on lines 3-18 can be iterated at most quadratic amount of times to n before terminating. On a single iteration the `UpdateTable` calls and the `UpdateVisited` call are linear to n, giving $\mathrm{O}(n^3)$ as the worst-case complexity of the algorithm.

- We rely on the correctness of the DFS algorithm to find all paths that do not rely on "undefined" values (as discussed in Section 5.3) of the individual procedure graph.
- We can also see that `AnalyzeProcedure` is monotonic over the Table, as it only modifies the Table through `UpdateTable`, which is trivially monotonic.

`ComputeTable`:

- Note that the partial order between tables is complete since the poset is finite. Since `AnalyzeProcedure` is monotonic over the Table the conditions for the Kleene fixed-point theorem (see Section 2.1.3) are fulfilled. Starting with the least element Table on lines 4-5 we get the least fixpoint of `AnalyzeProcedure` w.r.t. the Table from the ascending Kleene chain by iteration of lines 7-16. The amount of iterations required for the chain to reach the supremum is the amount of iterations that the Table can be modified, which is in worst case quadratic to n, times the maximal amount of elements that can be added to Worklist on a single iteration, which is linear to n. On a single iteration, `AnalyzeProcedure` is run in $\mathrm{O}(n^3)$ and it majors other operations of the iteration. Therefore `ComputeTable` runs in $\mathrm{O}(n^6)$.
- The correctness of the algorithm requires that when a fixpoint is reached all possible paths of the graph are found and therefore the minimal path lengths that the Table contains are the shortest path lengths. To see why this is true note that with a correct `AnalyzeProcedure` implementation, the only way in which paths are missed is since they depend on "undefined" values as shortest paths to return nodes. The only way in which the final result can have such values is to have a cycle of procedures calling each other, and shortest paths do not contain cycles.

`SPCv1` and `SPCv2`:

- For the first part of the `SPCv1` and `SPCv2` algorithms, `UpdateVisited` limits the amount of iterations to $|S| \cdot n^2$. With each iteration requiring a call to `UpdateVisited`, the total running time becomes $\mathrm{O}(|S|^2 \cdot n^3)$ for both algorithms for cases where `ComputeTable` is not called and in $\mathrm{O}(|S|^2 \cdot n^6)$ for cases where it is called. Both `SPCv1` and `SPCv2` are in the same complexity class. We leave for the empiria to decide which suits our task better.
- For correctness, we rely on the arguments that were given while constructing the algorithms.

Our method of considering worst-case execution times gives an unnecessarily pessimistic picture of the performance of the algorithms. For an example, in practice it is seldom that a path length is anywhere near the theoretical maximum path length. Amortized analysis could possibly give a better picture, but such an analysis is out of our scope. Finally, for memory requirements, the containers used take elements corresponding to unique nodes or unique procedures, giving a rough $\mathrm{O}(n)$ (with the exception of Visited that can take a stack too, giving $\mathrm{O}(|S| \cdot n)$).

Chapter 6

Experimental Results

In this chapter we give the results obtained by running Conformiq Qtronic's test generation algorithm on nine industrial models with and without our design.

6.1 Test Setup

The tests were run on Conformiq Qtronic development version 2.1.2+. The software is proprietary but a fully working evaluation version can be obtained from the company website [1]. The public version at the time of writing is 2.1.2 and it was determined that the execution times of the test generation algorithm have not significantly changed from 2.1.2 to the development version 2.1.2+.

The tests were run between the 26th and the 28th of February 2010.

The test machine had the following specifications:

- Inter Core i7-920 2,66 GHz processor (quad-core),
- 6 GB 1600 MHz DDR3 memory,
- Linux 2.6.28, and
- GCC 4.3.3.

The processor is quad-core, but the test generation algorithm was run with one process and only one core was allocated to the task.

The tests were run by compiling and executing the source files with ten different configurations of options. The options were:

- running the base symbolic execution based test generation algorithm (abbreviated BA),
- running the flow analysis algorithm (FA),
- doing correspondence tracking between the execution of the symbolic execution and the flow analysis PDA (CT),

- running version 1 or version 2 of the shortest path calculation algorithm (SPCv1 and SPCv2, respectively),
- cutting symbolic execution state space search (Cut),
- having access to symbolic execution's dynamic target checkpoint information (CPInfo), and
- having access to symbolic execution's dynamic bound information (BInfo).

The configurations were grouped into four groups:

1. Firstly, to form a base case to which other configurations can be compared to the base symbolic execution test generation algorithm was run only.
2. Secondly, to give an answer to our main question of whether our design can reduce test generation time or not the whole system was run and the results compared to the base case. Separate runs for both versions of the shortest path calculation algorithm were run to determine which algorithm suits the task better.
3. Thirdly, there was a need to confirm that using flow analysis alone cannot cut the search as well as with shortest path calculation. Cutting the search with flow analysis only corresponds to cutting the rest of the branches of the search space when all the checkpoints in the flow analysis PDA have been reached. Also, the impact of incomplete information to the system's performance was checked by running the whole system but omitting the symbolic execution's dynamic checkpoint or bound information for the shortest path calculation algorithm.
4. Lastly, to determine the overhead of our system's components separate runs were organized so that we only had a part of the components switched on at a time. Because of the dependencies of the components this had to be done so that we had a run of the base test generation and flow analysis algorithms only, then a run of the same components with the addition of correspondence tracking, and then runs of the same components with the additions of correspondence tracking and both versions of the shortest path calculation algorithms. Because of the small granularity of calls to correspondence tracking and shortest path calculation we measured the time taken by the components by running the system with the component switched on and reducing from this time the corresponding running time of the system without the component. Since search space cutting was not switched on, we might have gotten a pessimistic view of the overhead since cutting the search could have reduced the amount of calls needed of correspondence tracking and the shortest path algorithm. With this setup, however, we could get an upper bound to the overhead.

The configurations and their options are summarized in Table 6.1, where 'x' denotes that the corresponding option was switched on in the corresponding configuration.

For each configuration a test run consisted of running the system on nine models. After a run the executables were compiled again and another run was instantiated. Each configuration was run in this way five times for redundancy and to reduce the effect of the possible natural variation in consecutive execution times.

Conf.	BA	FA	CT	SPCv1	SPCv2	Cut	CPInfo	BInfo
1	x						x	x
2a	x	x	x	x		x	x	x
2b	x	x	x		x	x	x	x
3a	x	x				x	x	x
3b	x	x	x		x	x		x
3c	x	x	x		x	x	x	
4a	x	x					x	x
4b	x	x	x				x	x
4c	x	x	x	x			x	x
4d	x	x	x		x		x	x

Table 6.1: Configuration options matrix.

6.2 Industrial Examples

6.2.1 Models

Our test set consists of nine models that model real-world systems. The Inventory Client, Inventory System and SIP Client models have been developed in-house, other models come directly from clients or from company consultants that have created the models with client contacts. Apart from the Inventory System model, which includes the Inventory Client model as a part of it, the models share no common structures outside of library code. The models have been developed without a connection to the development of flow analysis or shortest path calculation. The author has had no input in the creation or development of these models.

We view that although the size of the test set could always be bigger, the selection gives enough insight to the system's performance in the common use cases of the product. We identify two areas where testing is mostly performed on: communication protocol and database-oriented services testing.

Table 6.2 gives a description of the systems the models model. The systems are often complex and the models describe just the fraction of the system that is relevant for testing, such as the message passing in a communication protocol (as opposed to e.g. message parsing functionality).

Table 6.3 gives some structural information of the models. Included are the amount of lines of code (LoC) in Java and the amount of UML diagram states and transitions that the model comprises of. Then, we have the lines of code of the resulting CQλ code and the amount of checkpoints that were inserted to the model according to the user's selected options. Lastly, we note if the models are multithreaded or not. The features do not directly describe how computationally demanding the models are (a small model can always contain a specially demanding structure such as a point of a lot of branching) but they give an overview of the sizes of the models.

BGP	A Border Gateway Protocol (BGP) router. The protocol is the core routing protocol of the Internet. The router for this protocol is a machine that maintains a routing table of items which designate network reachability among systems connected to the Internet.
BootTest	The boot sequence of an unspecified router and the associated filesystem checks needed to perform the booting operation. The router can be booted from a number of file formats and devices such as a flash drive.
TwoRouters	Two interconnected routers that negotiate their settings by using Internet Control Message Protocol and Simple Network Management Protocol (SNMP). The internal logic of the protocols is abstracted out of the model and the focus is on correct protocol message passing.
SCTP	The Stream Control Transmission Protocol (SCTP) with the focus on the handling of SCTP ports and the distribution of them to participants in the protocol.
Instiki	The Instiki Wiki clone engine with a simple accompanying database.
Inventory Client	The client side of a client-server architecture item warehouse application.
Inventory System	As an extension of Inventory Client, the case models the whole client-server architecture item warehouse application. The system consists of a client, a server and a database that the server uses. The client can poll information on the items in the warehouse or send a flag and receive updates automatically.
NoTA	Two subsystems on a Network on Terminal Architecture (NoTA). NoTA is a modular service-based architecture for mobile and embedded devices. A subsystem corresponds to a physical implementation of a device in this architecture.
SIP Client	The client side of the Session Initiation Protocol (SIP), focusing on session initiation and abstracting optional elements out.

Table 6.2: Descriptions of test case models.

	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
Java LoC	268	271	303	2766	323	271	496	1423	130
UML States	8	20	2	56	27	10	12	25	23
UML Trans.	16	22	5	75	31	16	22	54	23
CQλ LoC	16486	14993	10227	123205	20775	12373	17442	51862	11088
Checkpoints	92	108	64	678	147	62	96	308	75
Multithreaded?	yes	yes	yes	no	no	no	yes	yes	no

Table 6.3: Structural features of test case models.

Conf. 1	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$5.5 \cdot 10^7$	$1.4 \cdot 10^6$	$3.1 \cdot 10^8$	$3.8 \cdot 10^7$	$6.8 \cdot 10^7$	$9.8 \cdot 10^6$	$4.1 \cdot 10^7$	$9.7 \cdot 10^8$	$3.3 \cdot 10^5$
sat queries	75249	1612	349374	28341	64795	12941	49157	707253	1620
avg time	**81.0**	**0.7**	**270.7**	**33.6**	**97.9**	**9.9**	**31.8**	**654.8**	**0.7**
max dev	1.0	0.0	2.9	0.3	1.3	0.1	0.2	4.8	0.0

Table 6.4: The base test generation algorithm (BA).

Conf. 2a	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$6.0 \cdot 10^6$	$1.3 \cdot 10^6$	$5.0 \cdot 10^6$	$3.8 \cdot 10^7$	$3.3 \cdot 10^7$	$1.8 \cdot 10^6$	$2.3 \cdot 10^7$	$9.7 \cdot 10^8$	$2.9 \cdot 10^5$
sat queries	8094	1513	7729	27893	32386	3456	30190	707253	1252
sp queries	2845	615	6001	14748	28958	2404	14560	318783	523
cut %	20.4	2.1	34.3	1.5	10.4	23.6	10.3	0	23.1
avg time	16.7	1.3	9.1	226.5	97.1	4.0	42.0	1133.4	0.7
max dev	0.4	0.0	0.0	0.5	1.6	0.0	0.6	5.0	0.0
% of BA	**20.7**	**201.2**	**3.4**	**674.5**	**99.2**	**40.1**	**132.1**	**173.1**	**105.0**
Conf. 2b	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$6.0 \cdot 10^6$	$1.3 \cdot 10^6$	$5.0 \cdot 10^6$	$3.8 \cdot 10^7$	$3.3 \cdot 10^7$	$1.8 \cdot 10^6$	$2.3 \cdot 10^7$	$9.7 \cdot 10^8$	$2.9 \cdot 10^5$
sat queries	8094	1513	7729	27893	32386	3456	30190	707253	1252
sp queries	2845	615	6001	14748	28958	2404	14560	318783	523
cut %	20.4	2.1	34.3	1.5	10.4	23.6	10.3	0	23.1
avg time	12.0	1.2	6.2	223.0	89.6	3.1	24.1	879.6	0.7
max dev	0.1	0.0	0.1	0.6	1.8	0.0	0.4	8.5	0.0
% of BA	**14.8**	**175.6**	**2.3**	**664.0**	**91.5**	**31.6**	**75.8**	**134.3**	**100.5**

Table 6.5: The runs of the whole system with SPCv1 (2a) and SPCv2 (2b).

6.2.2 Results

Tables 6.4 (the base case), 6.5 (comparisons to the whole system), 6.6 (comparison to cutting search with flow analysis only) and 6.7 (component overheads) give the results of the test runs. The tables are organized by a number of attributes. We record the amount of execution steps and the amount of satisfiability queries the symbolic execution system takes during its search (vm steps and sat queries, respectively). The time taken by the execution is averaged over five runs (avg time) and the maximal absolute deviation from this average (max dev) is also measured. Both of these times are given in seconds. On configurations where the search space is cut we compare the execution time to the base case (% of BA) and we record the amount of shortest path calculation queries (sp queries, corresponding to the amount of branches in the model) and the percentage of queries where the branch could be cut (cut %). The emphasis is on the test generation time of the base algorithm and its comparisons to the whole system and the system's components.

We give an analysis on the results in Chapter 7.

6.3 In-Depth Example

To give a more concrete picture of how the system affects the efficiency of the test generation algorithm we now look at one model in a bit more detail. The model chosen here is Inventory System.

Conf. 3a	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$5.1 \cdot 10^7$	$1.4 \cdot 10^6$	$2.9 \cdot 10^8$	$3.8 \cdot 10^7$	$6.8 \cdot 10^7$	$8.8 \cdot 10^6$	$4.1 \cdot 10^7$	$9.7 \cdot 10^8$	$3.3 \cdot 10^5$
sat queries	70681	1612	323842	28341	64795	11490	49157	707253	1620
avg time	85.4	1.1	262.9	212.7	121.2	8.2	36.3	782.0	0.9
max dev	0.9	0.0	1.1	0.3	0.4	0.1	0.5	5.6	0.0
% of BA	**105.4**	**168.4**	**97.1**	**633.4**	**123.8**	**83.1**	**114.1**	**119.4**	**125.2**
Conf. 3b	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$5.5 \cdot 10^7$	$1.4 \cdot 10^6$	$3.1 \cdot 10^8$	$3.8 \cdot 10^7$	$6.8 \cdot 10^7$	$9.8 \cdot 10^6$	$4.1 \cdot 10^7$	$9.7 \cdot 10^8$	$3.3 \cdot 10^5$
sat queries	75249	1612	349374	27896	61209	12301	49157	707253	1506
sp queries	38084	681	231062	14748	50868	7504	22264	318783	699
cut %	0	0	0	1.5	6.5	4.5	0	0	16.2
avg time	92.8	1.2	289.6	213.9	123.8	11.5	37.1	802.5	0.9
max dev	0.9	0.0	2.2	0.3	0.6	0.2	0.3	5.5	0.0
% of BA	**114.6**	**175.7**	**107.0**	**636.9**	**126.5**	**116.4**	**116.5**	**122.6**	**125.0**
Conf. 3c	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
vm steps	$6.0 \cdot 10^6$	$1.3 \cdot 10^6$	$5.1 \cdot 10^6$	$3.8 \cdot 10^7$	$6.7 \cdot 10^7$	$7.1 \cdot 10^6$	$4.1 \cdot 10^7$	$9.7 \cdot 10^8$	$3.1 \cdot 10^5$
sat queries	8094	1513	7813	27935	59999	8746	48015	707253	1351
sp queries	2845	615	6001	14758	50764	5367	22264	318783	594
cut %	20.4	2.1	34.0	1.4	7.7	6.7	4.2	0.0	20.9
avg time	12.1	1.2	6.3	223.0	145.8	7.5	42.4	881.0	0.8
max dev	0.1	0	0.1	0.4	1.3	0.2	0.2	4.8	0.0
% of BA	**14.9**	**175.8**	**2.3**	**664.0**	**149.0**	**76.0**	**133.4**	**134.6**	**110.0**

Table 6.6: Cutting search with FA only (3a) and the impact of incomplete information (3b,3c).

Conf. 4a	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
avg time	6.3	0.4	0.5	175.2	16.4	1.1	2.4	80.3	0.2
max dev	0.1	0.3	0.0	2.0	0.2	0.0	0.0	0.5	0.0
% of BA	**7.7**	**62.5**	**0.2**	**521.7**	**16.8**	**10.7**	**7.5**	**12.3**	**23.5**
Conf. 4b	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
avg time	5.2	0.0	16.0	4.1	7.0	0.5	2.1	51.6	0.0
max dev	0.9	0	2.8	0.4	1.1	0.1	0.2	9.9	0.0
% of BA	**6.4**	**6.7**	**5.9**	**12.2**	**7.1**	**5.4**	**6.7**	**7.9**	**1.9**
Conf. 4c	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
avg time	313.3	0.4	482.7	13.8	195.4	11.4	46.8	336.4	0.2
max dev	1.0	0.0	10.4	0.3	5.3	0.3	1.2	15.4	0.0
% of BA	**386.8**	**63.6**	**178.4**	**41.1**	**199.6**	**114.8**	**147.3**	**51.4**	**21.1**
Conf. 4d	BGP	BT	TR	SCTP	Instiki	ICli	ISys	NoTA	SIP
avg time	7.3	0.2	93.2	10.4	28.4	1.2	6.7	94.1	0.1
max dev	1.2	0.0	6.5	0.1	1.2	0.1	0.2	5.3	0.0
% of BA	**9.0**	**20.8**	**34.4**	**31.0**	**29.0**	**12.5**	**21.1**	**14.4**	**13.5**

Table 6.7: Component overheads, FA(4a), CT(4b), SPCv1(4c), SPCv2(4d).

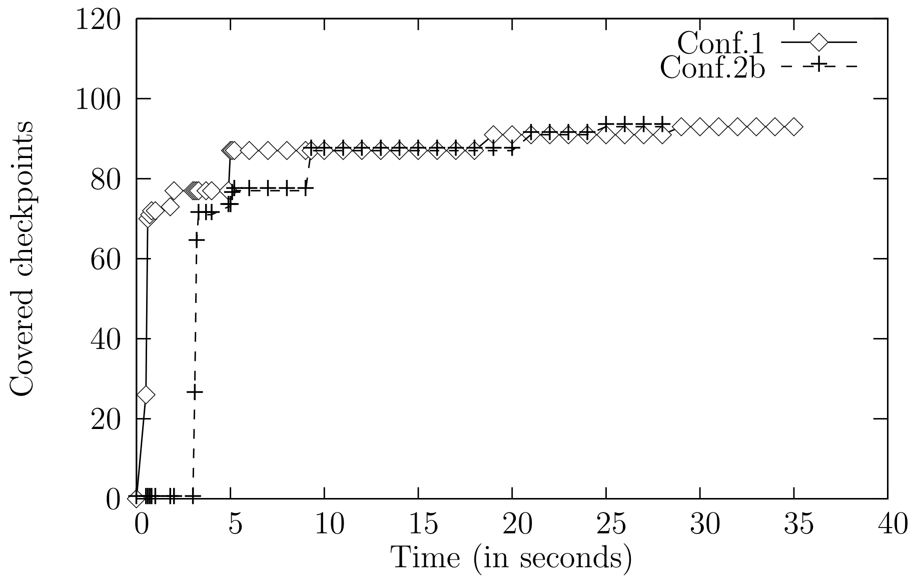

Figure 6.1: Covered checkpoints over time in the Inventory System model.

First, in Figure 6.1, we show how the state space search of the test generation algorithm covers target checkpoints over time. The line pointed with diamonds corresponds to the base algorithm and there we can see how the majority of the target checkpoints get covered in a short space of time (most of the checkpoints that will ever be covered are covered already after five seconds). After the initial surge finding more checkpoints becomes increasingly harder, which is easily attributed to the state explosion problem (increasing path length increases the amount of possible states exponentially). The search finally concludes at around 36 seconds. The line pointed with plus marks corresponds to the run of the whole system with shortest path calculation algorithm version 2. Here, the first three seconds are spent on flow analysis. After that the search proceeds by cutting branches where possible, finally "catching" the base algorithm at around 22 seconds. The likelihood of cutting branches increases when more and more targets are already covered and thus the search is terminated at around 28 seconds, in circa 8 seconds less total time than the base algorithm.

As an example of non-trivial branches that the system cuts from the search, consider Figures 6.2 and 6.3 that contain the UML portions of the Inventory System model's client and server sides, respectively. Here, an execution state maps both threads to a state in the diagram, e.g. the initial state is mapped to the "Manual update mode" state for the client and the "Process messages" state for the server. State traversing is done by triggering transitions, which contain the type of the triggering event and the (possibly empty) action block containing code that is run when the transition is taken. The procedures pointed by the code are implemented in the Java portion of the model and do not matter in the context of this example. The "input"-typed triggers correspond to external input and "receive"-typed triggers correspond to receiving input through communication between the threads.

Now, during the symbolic execution of the model, the search proceeds to a situation where there are only a few checkpoints that have not been covered yet. All of these

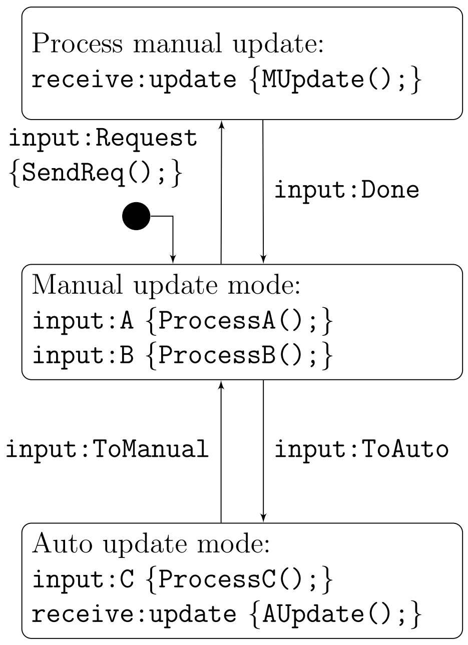

Figure 6.2: Client side of the Inventory System model.

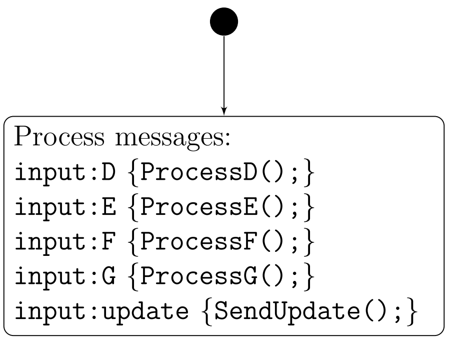

Figure 6.3: Server side of the Inventory System model.

checkpoints lie in the client side AUpdate procedure code. To reach these checkpoints with bound 1 client must be on the state "Auto update mode" and server must be on the state "Process messages" and the "input:update" transition in server must be triggered. Selecting any other "input"-typed transition to trigger (i.e. selecting another branch in the search space) will pass through a stable state and make the bound 0 without covering new checkpoints thus dropping the search based on the bound. Now, the system can direct symbolic execution to the correct branch by cutting search on all branches that do not correspond to being on the correct states and taking the right transition, e.g. client being on some state other than "Auto update mode" or server taking the "input:E" transition. If the bound is 2, there are a few more branches from which the checkpoints can be potentially reached from, but here also the execution must either already be on the correct states or reach them by going through maximally one stable state, which still leaves searching through many state configurations unnecessary, e.g. all states where the client is on the "Process manual update" state.

6.4 Problematic Structures

For completeness, it needs to be noted that the Inventory System model contains one structure that at first had a degrading effect on the system. The structure is found in the third thread of the model, which models a database that handles requests given to it by the server and answers in a suitable way. The basic loop of the database thread is given in Figure 6.4. During translation to CQλ checkpoints are inserted to the if-else if-else construct on lines 9-17. Now, it should be clear that the implicit else

```
 1: ...
 2: while (true)
 3: {
 4:   Message msg=receive();
 5:   assert(
 6:    msg instof Msg1 ||
 7:    msg instof Msg2
 8:   );
 9:   if (msg instof Msg1)
10:   {
11:     ...
12:   }
13:   else if (msg instof Msg2)
14:   {
15:     ...
16:   }
17:   [implicit else branch]
18: }
19: ...
```

Figure 6.4: Trivial problematic structure in the Inventory System model.

```
 1: ...
 2: public foo(x)
 3: {
 4:   [complex processing]
 5: }
 6: ...
 7: public bar(x)
 8: {
 9:   assert(foo(x) == 1);
10:   if (x == 0)
11:   {
12:     [reachable?]
13:   }
14:   else
15:   {
16:     [reachable?]
17:   }
18: }
19: ...
```

Figure 6.5: Non-trivial problematic structure.

branch on line 17 cannot be reached since it has been asserted that a message received (from server) is one of two types and the types are already considered. However, here the flow analysis is (due to implementational imprecision) unable to determine so and overapproximates the possible branches by outputting a checkpoint for the implicit else branch into the PDA graph. Since receiving a message from another thread does not constitute a stable state the checkpoint in the PDA graph is reachable with any bound and since it cannot be covered in reality it stays as a target during the whole symbolic execution state space search. This prevents the system from cutting any branches.

Such a sensitivity to unreachable checkpoints is not desirable. For trivial cases such as the one described here it sufficed to use a handling separate to flow analysis to disallow such checkpoints from appearing in the PDA graph. This handling removed a total of 1 checkpoint from the BGP and the Inventory System models, 2 checkpoints from the TwoRouters model and 4 checkpoints from the Instiki model. Ideally, we would like our base flow analysis algorithm to be precise enough so that this operation would not be required.

We also encountered non-trivial cases of the same problem in the SCTP and the NoTA models. An example of these is given in Figure 6.5 where the reachability of the if-else construct branches on lines 10-17 depend on the value of x. The assertion on line 9 maps x to a return value by complex processing and so can place arbitrarily complex requirements on the value of x. Accurately determining the possible values for x after the assertion would require a decision procedure such as used by symbolic execution in the worst case. The non-trivial cases encountered fall into two categories: the ones such as described just now and multithreaded communication precision issues.

Chapter 7

Analysis

This chapter gives an analysis of the results obtained in the last chapter, discusses the extent to which they are applicable in similar systems and considers alternative approaches.

The original research problem was given in the introduction as the following: can symbolic execution time, and therefore test generation time, be substantially reduced by determining an upper bound to the possible execution paths of the given model with flow analysis, and, during symbolic execution, identifying (and cutting symbolic execution search on) some of the paths from which expansion of the symbolic execution search would not find any new marked syntactic control flow points (i.e. checkpoints) within the given bound?

The problem was also divided into three subproblems since, in the worst case, the running time of the symbolic execution algorithm cannot be affected positively (see Chapter 1). We have now identified more free variables in the problem - most prominently that decisions related to the precision/efficiency tradeoff in both the symbolic execution system and the flow analysis system can affect the results considerably.

Therefore we base our analysis on answering the three subproblems by looking at possible trends in our results. Recapping, the subproblems are:

1. How substantial can the improvement in test generation times be in cases where the symbolic execution search is cut substantially by the branch cutting system?
2. How substantial can the induced overhead of the branch cutting system be in cases where the symbolic execution search is not cut substantially?
3. Is it more common that the branch cutting system is able or that it is not able to cut symbolic execution search substantially?

7.1 Effect on Test Generation Times

We note first the results of the base symbolic execution algorithm that are given in Table 6.4. In addition to comparing execution times we can compare the amount of steps in search and the amount of calls to a decision procedure.

Table 6.6 (Conf. 3a) shows how cutting the search based on flow analysis only (without shortest path calculation) affected execution times. We see a circa 20% decrease in execution time on the Inventory Client model but otherwise the times increase due to the overhead. We can therefore say that although flow analysis in itself can improve test generation times in some cases, it also induces at least the same amount of overhead in the cases where it does not cut the search effectively and most importantly it is most often not able to improve execution times.

Before looking at the effect of the whole system on execution times we compare the performance of the two versions of the shortest path calculation algorithm. In Table 6.5 we can affirm that both versions of the algorithm give the same results on queries but also that the execution times are smaller with version 2. Also, looking at the upper bounds of the induced overheads of the algorithms in Table 6.7 (Conf. 4c and 4d), we can see that the amounts for version 2 are (in some cases dramatically) smaller than for version 1. It seems that the set of target checkpoints changes seldom enough to warrant tabularization on each change. This can be attributed to some extent to that although the amount of checkpoints is linear to model size the amount is in practice (Table 6.3) small compared to the amount of queries to the shortest path algorithm (Table 6.5). We conclude that the version 2 of the algorithm performs consistently better than version 1 in our problem instances and only consider results related to version 2 from now on.

Looking at Table 6.5 (Conf. 2b) we can see that running the whole branch cutting system decreased test generation times into the following fractions of the original times on five models: 91.5% on Instiki, 75.8% on Inventory System, 31.6% on Inventory Client, 14.8% on BGP and 2.3% on TwoRouters. There was no substantial change on the time on SIP Client model. Three models saw an increase in test generation time: 34.3% on NoTA, 75.6% on BootTest and 564% on SCTP.

At first the results seem rather inconclusive. The results range from a dramatic decrease to 2.3% to a dramatic increase to 564% of original times. However, we can conclude that the improvement in test generation times can be very substantial in cases where the system is able to cut the search.

To get a better view on how substantial the induced overheads can be we look at the individual component overheads in Table 6.7. Flow analysis (Conf. 4a) shows overheads that lie generally below circa 20% of the base symbolic execution algorithm time. There are two spikes with BootTest taking 62.5% and SCTP taking 521.7% of the base algorithm's time. These are also the two models for which the whole system causes the most increase in test generation time. If the overhead of flow analysis would be reduced to below 20% of base algorithm's time with these cases the overall overhead would be reduced to circa 35% on BootTest and circa 60% on SCTP. Looking at the implementation of flow analysis we are confident in that the spikes are the result of prototype implementational issues and not a sign of scalability issues. The fact that the flow analysis overhead on the structurally comparatively large NoTA model (Table 6.3) is circa 12% gives some credence to this claim. The overhead of correspondence tracking is (unsurprisingly) negligible (Conf. 4b). The upper bound on the overhead of the shortest path calculation component (Conf. 4d) stays below 35% of the base algorithm's time, with the size of model structure not having any apparent effect. This gives a clue of the algorithm's scalability, although a separate scalability test with a larger test set would be required to confirm scalability. Based on the theoretical and measured scalability as well as the measured actual amounts we can conclude that the induced overhead of the system is not as substantial as the possible improvements measured in the previous paragraphs.

As to whether it is more common for the system to be able to cut search or not, we can look at the cut percentages of Table 6.5 (Conf. 2b). We see that the percentage is noticeable (over 10%) on six cases (five of which had improvements in test generation time) and small or non-existent (less than 2.5%) on the three cases which saw increased test generation time. In addition, four of the "good" cases initially encountered trivial problematic constructs (Section 6.4) and the "bad" cases encountered non-trivial problematic constructs. The common factor for the problematic constructs were that due to imprecision flow analysis generated false positive checkpoints for them to the control flow graph which in some cases prevented search cutting. It is important to note that a majority of our cases encountered these constructs and that non-trivial constructs were encountered on structurally large models. It is likely that with larger models non-trivial constructs will be encountered even more - placing demands that either the result of flow analysis must be refined or the false positives must be picked out by another system.

To summarize, on the upside the branch cutting system can improve test generation times substantially (even dramatically) and the induced overhead of the system is tolerable but on the downside the system is vulnerable to false positives of the flow analysis.

7.2 Applicability of Results

The branch cutting system can be applied to any symbolic execution system for which the search objective is to reach each search target at least once (instead of verifying properties through all paths).

Any static analysis method can be used, the essential part is that the method needs to calculate a safe overapproximation of the control flow graph into a pushdown automata form and do an precision/efficiency tradeoff based on the corresponding precision/efficiency tradeoff done in the symbolic execution system.

The system requires the minimal interface of the control flow graph represented as a pushdown automaton, the bounding heuristic representable as edge weights of the graph, the current state of the symbolic execution search represented as a branch signature, the current set of targets and the current bound heuristic counter value.

The precision/efficiency tradeoff decisions in both the symbolic execution system and the flow analysis system and the application domain from which the input models come from can affect the results. Therefore we claim that numerically our results are valid mostly in communication protocol and database-oriented services application domains. We however also claim that the more general conclusion that our system can improve test generation times substantially (as long as false positives do not prevent its usage) is valid through multiple domains.

7.3 Alternative Approaches

We have only considered here flow analysis systems that produce a safe overapproximation of the semantics of programs. Underapproximative flow analysis, i.e. flow analysis that uses underapproximative abstractions, is also possible [4]. Of particular interest to us is the possibility of using underapproximative abstractions in conjunction with overapproximative ones to allow for faster flow analysis as well as removing some problematic false positives from the control flow graph. This would, however, come with the cost of making our system cut search on some branches for which symbolic execution would find targets within the bound.

We have not focused much on the memory requirement of our system. The control flow graph is the main memory-consuming artifact of our system and as an alternative way of doing flow analysis [42] presents a graph-free approach with a claim of circa 30% decrease in memory consumption. However, utilizing this approach would require us to accommodate for the lack of an explicit CFG and instead operate on the maximal fixed point data-flow analysis solution to determine shortest paths. We think that it is likely that some form of an explicit CFG would be required.

Our shortest path calculation component takes as inputs information about the current execution state of symbolic execution and answers as an output whether the execution should proceed from the current state. Therefore it can be viewed as a decision procedure with the following relation to decision procedures of symbolic execution: both our component and a symbolic execution decision procedure can be used to cut the search on current branch and our component runs in polynomial-time whereas a symbolic execution decision procedure is in the general sense exponential-time but our component only gives useful information when it answers that the search can be cut whereas a symbolic execution decision procedure is able to give data values that can be used to generate tests on valid actual execution paths.

We have then the possibility of deciding which of the decision procedures we want to run, or run first, to check whether execution should proceed at a given point. Usually we could use our general approach of running our component first and then either run or delay running the other. However, as seen in Figure 6.1, our system's ability to "compress search" (i.e. cut search branches) improves as more targets are covered so it could be more effective to start using our component first at some "later stages" of the search.

Chapter 8

Conclusions

8.1 Summary

Automated model-based test generation via symbolic execution of models written in a Scheme-like language was discussed. The task of the symbolic execution system in question is to cover the given model by finding concrete execution paths from the initial state of the model to target checkpoints (i.e. marked syntactic control flow points) within a given bound. Since it is enough to find at least one execution path to each checkpoint the symbolic execution search can be optimized by identifying (and cutting from the search) execution paths that would not reach new checkpoints within the given bound. The main product of the thesis is the design and implementation of a "branch cutting system", a system that does such an identification on a subset of the execution paths. The branch cutting system consists of the following parts:

1. A separate static analysis phase of flow analysis is run to compute a safe over-approximation of the control flow of the program. The analysis is implemented as a polynomial-time control flow analysis which produces a control flow graph (CFG) encoded as a pushdown automata -equivalent graph.
2. During symbolic execution current states are mapped to the CFG.
3. A pushdown automata shortest path calculation algorithm is used to determine whether new target checkpoints can be found within the given bound in the CFG and if not the symbolic execution search is cut on the current execution path. The shortest path calculation algorithm is implemented as a polynomial-time algorithm that tabulates partial shortest paths.

The performance of the branch cutting system was quantified empirically with an industrial model-based test generation tool Conformiq Qtronic [1]. Running the tool on nine industrial models resulted in the following findings:

1. For models for which no problematic non-trivial false positive target checkpoints were encountered in the flow analysis produced CFG the branch cutting system reduced test generation time substantially (to a fraction of circa 90% to even 3% of original times).

2. The induced overhead of the system was found to be tolerable (below 20% of the symbolic execution time for flow analysis and below 35% for shortest path calculation), with two spikes in the overhead of flow analysis that are with some confidence just artifacts of a suboptimal prototype implementation.
3. The branch cutting system was found to be very sensitive to false positive target checkpoints in the CFG making the precision of flow analysis imperative to the technique's applicability.

8.2 Future Work

We have established that the branch cutting system can reduce test generation time substantially when it is not hampered by false positives in the flow analysis result. Therefore (in our opinion) the imperative future goal will be to inspect ways of making flow analysis more precise.

The *counterexample-guided abstraction refinement* (CEGAR) principle [15] is a promising approach to improve precision. In CEGAR a coarse abstraction of a program is iteratively refined by adding more information about program states in cases where the original abstraction has been found imprecise. Imprecision is identified by checking the counterexamples reported by the abstraction. In our application this could mean e.g. that the spurious paths in the flow analysis control flow graph would be dropped when the corresponding paths have been found invalid by symbolic execution (a similar idea with symbolic simulation is explored in [32]) or that flow analysis would be iterated by using (as necessary) *relational domain abstraction* [44] i.e. data domains where relations of variables (e.g. $x \leq y$, $x \neq y$) are taken into account. CEGAR has been applied successfully in [8, 9].

Another source of imprecision that needs to be worked on is the field of issues arising from multithreadedness, e.g. shared data values that depend on thread scheduling. However, we realize that this is another large effort since the static analysis of multithreaded programs is undecidable in general [51].

We have considered improving symbolic execution with static analysis means. Others [8, 9, 32] have considered improving static analysis with symbolic execution (or simulation). The underlying duality of the two program analysis methods gives plenty of possibilities in that they can be used to narrow each other's approximated results.

Although the shortest path calculation algorithm is not the main bottleneck of our system, the efficiency of the algorithm could still be improved with further usage of incremental information. In particular, we could utilize the information given to us when the algorithm answers that there is a target within the given bound. The answer holds true as long as the symbolic execution system is executing through the path that was found and therefore the amount of calls to the algorithm could be reduced. Currently we only use the algorithm as a decision procedure. However, as discussed in [46], augmenting a decision procedure algorithm to answer to the corresponding search problem - i.e. to produce the actual shortest path in our case - does not essentially increase the computational complexity of the algorithm.

Bibliography

[1] Conformiq Software website. Online: `http://www.conformiq.com/` (visited on April 26, 2010).

[2] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. "Revised^5 Report on the Algorithmic Language Scheme". *Higher-Order and Symbolic Computing*, 11(1):7–105, 1998.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[4] S. Anand, C. S. Păsăreanu, and W. Visser. "Symbolic Execution with Abstraction". *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):53–67, 2009.

[5] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. "Zing: Exploiting Program Structure for Model Checking Concurrent Software". In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR), London, UK, August 31 - September 3, 2004*, pages 1–15.

[6] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[7] J. M. Ashley and R. K. Dybvig. "A Practical and Flexible Flow Analysis for Higher-Order Languages". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):845–868, 1998.

[8] T. Ball and S. K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis". In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon, USA, 2002*, pages 1–3.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. "The Software Model Checker BLAST: Applications to Software Engineering". *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 2007.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs". In *Proceedings of the 5th International Conference on Tools and*

Algorithms for Construction and Analysis of Systems (TACAS), Amsterdam, The Netherlands, March 22-28, 1999, pages 193–207.

[11] R. S. Boyer, B. Elspas, and K. N. Levitt. "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution". *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[12] G. Brat, K. Havelund, S. Park, and W. Visser. "Java PathFinder - Second Generation of a Java Model Checker". In *Proceedings of the Workshop on Advances in Verification, Chicago, Illinois, July 2000.*

[13] C. Cadar, D. Dunbar, and D. R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, California, USA, December 8-10, 2008*, pages 209–224.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: Automatically Generating Inputs of Death". In *Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, 2006*, pages 322–335.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *Journal of the ACM*, 50(5):752–794, 2003.

[16] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Progress on the State Explosion Problem in Model Checking". In *Informatics - 10 Years Back. 10 Years Ahead., R. Wilhelm, ed., Lecture Notes in Computer Science, vol. 2000, Springer-Verlag, 2001*, pages 176–194.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[18] E. M. Clarke and J. M. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[19] L. A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs". *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[20] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms (2nd Edition).* McGraw-Hill Higher Education, 2001.

[21] A. Cortesi. "Widening Operators for Abstract Interpretation". In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Cape Town, South Africa, 10-14 November, 2008*, pages 31–40.

[22] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), Los Angeles, California, 1977*, pages 238–252.

[23] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. "Model-Based Testing in Practice". In *Proceedings of the 21st International Conference on Software Engineering (ICSE), Los Angeles, California, USA, 1999*, pages 285–294.

[24] M. Das, S. Lerner, and M. Seigle. "ESP: Path-Sensitive Program Verification in Polynomial Time". In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, 2002*, pages 57–68.

[25] R. Diestel. *Graph Theory (Graduate Texts in Mathematics) (3rd Edition)*. Springer, 2005.

[26] M. Dowson. "The Ariane 5 Software Failure". *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[27] V. D'Silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, 2008.

[28] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[29] J. Edvardsson. "A Survey on Automatic Test Data Generation". In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linköping (ECSEL), October 1999*, pages 21–28.

[30] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed Automated Random Testing". *ACM SIGPLAN Notices*, 40(6):213–223, 2005.

[31] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.

[32] H. Hampapuram, Y. Yang, and M. Das. "Symbolic Path Simulation in Path-Sensitive Dataflow Analysis". In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Lisbon, Portugal, September 5-6, 2005*, pages 52–58.

[33] T. Hansen, P. Schachte, and H. Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries". In *Proceedings of the 9th International Workshop on Runtime Verification (RV), Grenoble, France, June 26-28, 2009*, pages 76–92.

[34] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[35] W. E. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System". *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[36] S. Jagannathan and S. Weeks. "A Unified Treatment of Flow Analysis in Higher-Order Languages". In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California, United States, 1995*, pages 393–407.

[37] J. Kinder, F. Zuleger, and H. Veith. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries". In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Savannah, Georgia, USA, 2009*, pages 214–228.

[38] J. C. King. "Symbolic Execution and Program Testing". *Communications of the ACM*, 19(7):385–394, 1976.

[39] W. Landi. "Undecidability of Static Analysis". *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[40] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. "Using Symbolic Execution to Guide Test Generation: Research Articles". *Software Testing, Verification and Reliability*, 15(1):41–61, 2005.

[41] N. G. Leveson and C. S. Turner. "An Investigation of the Therac-25 Accidents". *Computer*, 26(7):18–41, 1993.

[42] M. Mohnen. "A Graph-Free Approach to Data-Flow Analysis". In *Proceedings of the 11th International Conference on Compiler Construction (CC), Grenoble, France, April 8-12, 2002*, pages 46–61.

[43] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[44] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., 1999.

[45] OMG. *Unified Modeling Language: Superstructure, version 2.1.1.* Object Modeling Group, February 2007.

[46] C. M. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[47] V. R. Pratt. "Anatomy of the Pentium Bug". In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT), Aarhus, Denmark, May 22-26, 1995*, pages 97–107.

[48] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc., 2005.

[49] C. Păsăreanu and W. Visser. "A Survey of New Trends in Symbolic Execution for Software Testing and Analysis". *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[50] S. Qadeer, S. K. Rajamani, and J. Rehof. "Summarizing Procedures in Concurrent Programs". *ACM SIGPLAN Notices*, 39(1):245–255, 2004.

[51] G. Ramalingam. "Context-Sensitive Synchronization-Sensitive Analysis is Undecidable". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416–430, 2000.

[52] K. Sen, D. Marinov, and G. Agha. "CUTE: A Concolic Unit Testing Engine for C". In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Lisbon, Portugal, 2005*, pages 263–272, 2005.

[53] O. Shivers. "Control Flow Analysis in Scheme". In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, 1988*, pages 164–174.

[54] O. Shivers. "The Semantics of Scheme Control-Flow Analysis". In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut, USA, 1991*, pages 190–198.

[55] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1996.

[56] S. S. Skiena. *The Algorithm Design Manual.* Springer-Verlag New York, Inc., 1998.

[57] W. Thomas. "The Reachability Problem Over Infinite Graphs". In *Proceedings of the Fourth International Computer Science Symposium in Russia (CSR), Novosibirsk, Russia, August 18-23, 2009*, pages 12–18.

[58] N. Tillmann and J. de Halleux. "Pex-White Box Test Generation for .NET". In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP), Prato, Italy, 2008*, pages 134–153.

[59] A. Valmari. "The State Explosion Problem". In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, Germany, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[60] A. Valmari. "Software Model Checking is a Rich Research Field". *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):1–11, 2009.

Index