Lehrstuhl für Informatik VII
der Technischen Universität München

Reachability in Pushdown Systems: Algorithms and Applications

Dejvuth Suwimonteerabuth

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Javier Esparza
	2. Prof. Dr. Ahmed Bouajjani, Universität Paris Diderot/Frankreich

Die Dissertation wurde am 27.01.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.05.2009 angenommen.

Kurzfassung

Diese Arbeit präsentiert Erreichbarkeitsanalysen für Pushdown-Systeme und ihre Anwendungen auf unterschiedliche Bereiche. Ein Pushdown-System ist eine stack-basierte Maschine, deren Stack unbegrenzt ist. Pushdown-Systeme sind ein natürliches Modell für sequenzielle Programme. Angeregt durch einige Anwendungen analysiert diese Arbeit das Erreichbarkeitsproblem in verallgemeinerten Pushdown-Modellen—alternierenden PushdownSystemen und Pushdown-Netzwerken—und diskutiert sie im Detail.

Ein Pushdown-Netzwerk kann zur Modellierung nebenläufiger Programme verwendet werden. Die Erreichbarkeitsalgorithmen hierfür, zusammen mit einem Übersetzer von Java-Bytecode in Pushdown-Netzwerke, wurden optimiert und in einem Tool namens jMoped implementiert. jMoped ist ein Eclipse-Plugin, das Benutzern das einfache Testen von Java-Programmen ermöglicht, ohne dass sie die dazu verwendeten Techniken verstehen müssen. jMoped gibt fortlaufend den bislang erreichten Grad der Abdeckung aus und entdeckt Fehler wie z.B. Verletzungen von Assertions. Die Arbeit berichtet über praktische Experimente mit jMoped.

Alternierende Pushdown-Systeme werden als nützliches Modell für Autorisierungs- und Reputations-Systeme vorgestellt, deren Fragestellungen sich auf Erreichbarkeitsprobleme in den Modellen reduzieren lassen.

Abstract

This thesis presents reachability algorithms for pushdown systems and their applications to different areas. Roughly speaking, a pushdown system is a stack-based machine whose stack can be unbounded, making it a natural model for sequential programs. Inspired by applications, the thesis analyzes reachability in more generalized pushdown models—alternating pushdown systems and pushdown networks—and discusses their complexities in detail.

A pushdown network can be used for modeling multithreaded programs. The reachability algorithms, together with a translator from Java bytecodes into pushdown networks, have been optimized and implemented in a tool called jMoped. jMoped is an Eclipse plug-in which allows users to easily test Java programs without any knowledge of the techniques behind it. jMoped progressively outputs coverability information and uncovers errors such as assertion violations. Several practical experiments with jMoped are reported.

Alternating pushdown systems are shown to be suitable models for authorization systems and reputation systems, where reasoning in the systems boils down to solving reachability in the models.

Acknowledgments

This thesis would not have been possible without the support of many people. I would like to express my deepest gratitude to my supervisor Prof. Javier Esparza for his invaluable assistance, guidance, patience, and for giving me the opportunity to conduct research in his group. Special thanks go to Stefan Schwoon. Without his knowledge and assistance this research would not have been successful. I am very grateful to Prof. Ahmed Bouajjani, Tayssir Touili, and Mihaela Sighireanu for their abundant help with the research. I would also like to convey thanks to SFB 627 Nexus, Universität Stuttgart, and Technische Universität München for providing the financial and organizational support. Many thanks to my colleagues, especially to Stefan Kiefer and Michael Luttenberger, for their endless help and the great working atmosphere.

I would like to acknowledge Prof. Prabhas Chongstitvatana, the supervisor during my undergraduate study, for his inspiring advice. My parents and my brother, their love, understanding, and encouragement have never been absent, no matter when. Yupadara Netprapa, her love and understanding complete my life. I feel indebted to many people, in particular friends, for their bottomless help. They all play a role in this thesis, although not mentioned here. I thank you all.

Contents

Chapter 1

Introduction

Pioneered by Clarke and Emerson [16] and Quielle and Sifakis [49] in the early 1980s, model checking has emerged as an automatic technique for verifying computer systems based on exhaustive exploration of the space of reachable states. Model checking consists of three main tasks: Modeling, specification, and verification [17]. Given a *specification*—usually in a temporal logic formula—and a mathematical *model* of the system under consideration, model checking *verifies* the validity of the formula for the model. When the formula does not hold, users are often provided with an error trace. The trace can be used as an counterexample to track down the root of the error. The system can then be repaired and model-checked again.

When systems are finite, it is possible to use *finite state machines* as models. A finite state machine is a directed graph, in which nodes represent states of the system, and edges represent transitions between states. Nodes are labeled with sets of *atomic propositions* with the convention that their values are true in the nodes they label. Specification logics are built upon the atomic propositions together with logical and temporal operators. In the last three decades, a number of researches has been focused on different types of logics, and verification algorithms have been developed.

Even when systems under consideration are finite, every model checking algorithm usually faces a combinatorial blow up of the state space, which can be a result of e.g. an asynchronous system where processes are performed independently. This fundamental problem is known as the *state explosion problem*. The problem limited the usage of model checking in its early days.

A well-known approach that attempts to overcome the problem is to represent finite state machines *symbolically*. The idea, made popular by McMillan in 1992 [42], made use of Bryant's *binary decision diagrams* (BDDs) [8]. States can be compactly represented when using BDDs, and together with symbolic model checking algorithms where states can be efficiently manipulated, it was possible to verify a larger number of states than what explicit-state algorithms were able to handle.

Finite state machines are suitable for modeling hardware systems and communication protocols because they naturally involve only finite numbers of states. Many techniques have been extensively and successfully applied to these areas since the history of model checking. Model checking for software, however, has bee less adopted. The reason is because of its expressiveness by nature, making even very simple problems undecidable. In fact, the problem of deciding whether two threads with recursive calls can reach given points are already undecidable. Many recent researches in the software community have been focusing on finding reasonable solutions to overcome this problem.

This thesis focuses on a type of infinite-state systems which are based on stacks. In automata theory, a *pushdown automaton* is a well-studied language acceptor that makes use of a stack. Given an input, pushdown automata choose a transition based on the current control state and the symbol on top of the stack. The transition can optionally manipulate the stack by popping off the top of the stack and/or pushing new symbols onto the stack. A word is accepted by a pushdown automaton if starting from its initial state and stack symbol as the only element on the stack, one of its final states are reachable.

A *pushdown system* is a pushdown automaton but taking into consideration only the transition system it can generate, not the language it recognizes. The resulting transition system has infinite number of configurations of the form (control state, stack content) as states. Transitions between configurations are defined by moves of the pushdown automaton, without taking the input alphabet into account. Pushdown systems have been considered as a natural model for sequential programs. Procedure calls, including recursive calls, are easily handled by using stacks. As a consequence, there is no need to impose a bound on procedure calls as in approaches that employ finite-state models. One of the main restriction is, however, that it can only model data of finite domains.

Model checking pushdown systems was first introduced by Burkart and Steffen in 1992 [9], where alternation-free μ-calculus was considered. In 1996, Walukiewicz presented a procedure for finding winners in pushdown games, and pointed out that the procedure can be used to solve the model checking problem for the whole μ-calculus. One year later, Bouajjani, Esparza, and Maler proposed a simple automata-theoretic approach for the alternation-free fragment of μ-calculus [6]. Notably, they also pointed out that the reachability problem as well as the model checking problem for linear temporal logic (LTL) is polynomial in the size of the pushdown system. The solution makes use of finite automata as a data structure for representing infinite configurations. Given a finite automaton representing a set of initial configurations, the algorithms find all reachable states by adding transitions into the automaton until it is *saturated*, i.e. no more transitions can be added. The saturated automaton represents all configurations that are reachable from any initial configuration in the set. Concrete algorithms and exact complexity analyses were given later in [21].

In 2001, Alur, Etessami, Yannakakis [1] and Benedikt, Godefroid, Reps [4] independently investigated a model of computation closely related to pushdown systems called recursive (resp. hierarchical) state machines (RSMs). In fact, RSMs and pushdown systems possess the same expressiveness and succinctness but with a slightly different representation: an RSM explicitly models procedures as "boxes" of nodes (program states), where an edge entering (resp. leaving) a box models a procedure call (resp. return). Recursions are allowed. Efficient reachability algorithms and algorithms for model checking linear-time logic on RSMs have been proposed in both papers. The algorithms have a slightly better complexity compared to translating an RSM into a pushdown system and applying the algorithms from [21]. For a more thorough comparison, see e.g. [53].

In principle, when modeling programs with pushdown systems program variables (of finite domains) can be directly encoded into control states and stack symbols, introducing transition rules where variable relations are taken into account. However, doing so would lead to an unavoidable blow-up in the number of variables. To alleviate the problem, a BDD-based approach was presented in [24], where variables were treated symbolically with BDDs instead of direct encoding in control states and stack symbols. The model was called *symbolic pushdown system*. Symbolic algorithms, which generalize the previous algorithms, were implemented in the tool Moped as a part of Schwoon's thesis [53]. Moped supports reachability analyses and LTL model checking on both symbolic pushdown systems and Boolean programs. It has been successfully applied to a system equivalent to 10,000 lines of code of device drivers written in C.

A generalization of symbolic pushdown systems, called *weighted push-*

down systems, was introduced in [54]. In weighted pushdown systems, each transition is equipped with a value from a *bounded idempotent semiring.* A reachability problem of given two configurations is essentially the problem of determining their meet-over-all-paths value. [51] discusses the generalized reachability algorithms and their applications to interprocedural dataflow analyses in great detail.

Model checking for *probabilistic pushdown systems* was presented in [23]. A probabilistic pushdown system is a pushdown system whose transitions are associated with probabilities. The random walk problem, i.e. the problem of determining the probability of reaching a given configuration from another given configuration, was shown to be decidable. In this thesis, we generalize the idea to the semiring domain.

Pushdown systems have been applied to authorization problems in [31, 54, 32]. Given a system containing principals, resources, and rules, an authorization problem is a problem of determining whether a principal is authorized to access a given resource. SPKI/SDSI framework [20] provides a public-key infrastructure that emphasizes naming and authorization in a distributed environment. Its concept is simple and intuitive, however its expressiveness is enough to represent a wide range of applications. In SPKI/SDSI, principals are represented by their public keys. There are two types are certificates: name certificates and authorization certificates. A name certificate provides a definition of a local name in the issuer's local name space. An authorization certificate grants or delegates an authorization. The authorization problem can be defined as follows: given name and authorization certificates, can a principal access a resource? The problem boils down to *certificate chain discovery*, which involves finding of relevant certificates in order to prove the access [15]. Jha and Reps first observed that name and authorization certificates can be interpreted as a pushdown system [31], therefore the authorization problem reduces to the problem of pushdown reachability and can be solved by using e.g. the algorithms from [6, 21].

1.1 Contribution of the thesis

The thesis can be seen as a continuation of the work of Schwoon [53]. His model checker Moped has been proved to be efficient for analyzing reachability on symbolic pushdown systems. However, its applications to program testing in real-life programming languages had not been realized due to lacks of a front-end and a support to multithreading programs. An attempt that applies Moped to test sequential Java programs was made with an implementation of a translator from Java bytecode to pushdown systems in the author's Master thesis [56]. The translator, nevertheless, had several deficiencies as it supported only basic features of Java without for instance dynamic object creations and virtual method calls. The translator was also text-based, and therefore was only suitable for users who are familiar with pushdown systems. For this reason, we aimed not only to improve the translator or to develop new algorithms that would allow testing possible for a large set of Java programs, but also to develop a user-friendly tool that everybody can easily use without requiring expertise in the area of model checking.

In short, the thesis investigates algorithms and applications of pushdown systems and their variants. Three different pushdown models are introduced in Chapter 2: pushdown systems, alternating pushdown systems, and pushdown networks. Alternating pushdown systems generalize pushdown systems such that each transition can change the systems not only from a configuration to another, but to a *set* of configurations. A run can be seen as a tree of computations. Reachability analyses are more complex as a result. Pushdown networks are sets of pushdown systems. A pushdown network can be used as a model for a multithreaded program, in which a thread is represented by a pushdown system. Communications between threads are achieved via global variables. Also introduced in Chapter 2 are semirings as well as weighted versions of all three pushdown models. To be used as weights, binary decision diagrams are introduced at the end of the chapter.

Reachability analyses for weighted pushdown models are discussed in Chapter 3. The analyses are divided into two parts based on semirings. The first part considers the case when the semirings are bounded and idempotent for all three weighted pushdown models. We start by presenting the forward reachability algorithm for weighted pushdown systems from [51]. Then, we give an exponential-time backward reachability algorithm for weighted alternating pushdown systems including a detailed complexity analysis. We also prove that the reachability problem for alternating pushdown systems is EXPTIME-complete. We observe that if alternating pushdown systems satisfy certain constraints, then the exponential complexity can be avoided. A slightly modified algorithm, which runs in polynomial time, is given. For weighted pushdown networks, it is well known that the problem of deciding reachability is undecidable. As a result, we only compute approximations by applying *context-bounded analyses.* The idea is that we impose a bound on communications between threads, and only consider reachable configurations within the bound. The reachability algorithm for pushdown systems is applied each time a thread is active. Then, all possible values of global variables must be passed to other threads. This obviously leads to exponential blowups in the number of possible values of global variables. We propose a new approach which tries to avoid the blowups. It is still exponential in the worse case, but tends to perform well in practice (see Chapter 5). The second part of Chapter 3 deals with the semirings that are not bounded and idempotent. Reachability analyses in this case boil down to solving systems of polynomial equations.

Chapter 4 discusses an application of pushdown systems and pushdown networks to the area of Java program testing. Given a Java program to be tested, the program is first compiled into a class file containing Java *bytecode*—the machine language of the Java virtual machine. Then, a pushdown model is constructed from the class file such that a reachability analysis of the model can be seen as a simulation as if it were executed by the Java virtual machine. Given an input range, one can think of the reachability analysis as an execution of all possible inputs in a single run, instead of executing each input one by one as in the virtual machine. This enables us to find out not only coverage areas but also all errors of the program inside in the range within a single execution. Coverage information is usually useful to get more insight into the program under test, because we would normally like to cover every part of the code. The chapter first introduces basics of Java virtual machine, Java bytecode, and later its translation to pushdown models. Several issues that arise when applying the algorithms are resolved at the end of the chapter.

The translator and the reachability algorithms are implemented in the tool jMoped. jMoped is an Eclipse plug-in which allows users to easily test Java programs. Users simply select a Java method where the reachability analysis should start. jMoped then searches for all reachable statements assuming that the method parameters can take any possible values (within given bounds). During the analysis, markers of different types and colors are shown in front of Java statements. Black and green markers indicate that the corresponding statements are not reachable and reachable, respectively. Other types of markers reveal errors in the program, e.g. assertion violations are pointed out by red markers. Chapter 5 reports on experimental results with jMoped. The first experiment compares two different implementations of semirings: BDDs and bit vectors. The scalability of jMoped is measured in the second experiment where two quicksort implementations are considered. jMoped is able to find a bug in an implementation, and verify the correctness of another version in the case of 1-bit arrays of length 24 within 2.5 hours. In the next experiment, a part of jMoped, which is a recursive sequential program, has been successfully tested by jMoped itself. On the multithreaded side, jMoped is able to automatically find bugs in a Windows NT Bluetooth driver and `java.util.Vector` class from the Java library.

Chapter 6 lists three other applications of pushdown models: authorization systems, reputation systems, and pushdown games. The authorization systems and reputation systems are based on the SPKI/SDSI framework. Also considered in both systems are the presence of so-called *intersection certificates*, which results in the systems that correspond to weighted alternating pushdown systems. The two systems differ in that in authorization systems, an access is granted when at least one certificate chain with enough rights is discovered. On the contrary, in reputation systems, trusts from different chains can add up to increase the level of trust of an principal. We employ two different reachability algorithms depending on the types of weights. Pushdown games are introduced at the end of the chapter. We propose a solution which solves the games by translating them to alternating pushdown systems, and then applying the reachability analysis.

Chapter 7 concludes the thesis and discusses some possible future works.

1.2 Related works

Software verifications have been an active research area in the past decades. Spin [29] is probably the most popular tool for the formal verification of distributed software systems. It supports on-the-fly LTL model checking with partial order reductions, making it scales well even on very large problem sizes. Spin has been successfully applied to many real-life applications such as control algorithms, data communication protocols, and operating systems. Although Promela—the input language of Spin—supports dynamic process creation, it is difficult to encode programs in Promela due to its lack of procedure calls and objects. Some contributions in this direction include dSpin [19]—an extension of Spin with dynamic structures, a translator from ANSI-C to Promela [28], and the first generation of Java PathFinder which translate Java to Promela [25].

To overcome the limitations of Promela, a custom-made model checker has been developed in later versions of Java PathFinder [58]. It follows Spin as an explicit-state software model checker, but works directly on Java bytecode level. Java PathFinder is able to handle all bytecode instructions, and hence allows the whole of Java to be model checked. It can search for deadlocks and unhandled exceptions such as NullPointerException and AssertionError as well as violations of user-provided properties. Recently, it was able to uncover injected bugs in a very large application involving 18 threads, approximately 125,000 states and millions of paths. Previously hosted by NASA, Java PathFinder has been open-sourced since 2005 [30]. jMoped has similar features to Java PathFinder in that it works on the Java bytecode level, and can be seen as a virtual machine that can symbolically execute bytecode instructions for a given inputs in a single run. The models behind them, however, are different. Using a stack-based model, jMoped is able to naturally handle method calls as well as recursions which can result in infinite number of states. On the other hand, communications between threads are more expensive in jMoped, and as a result makes it unsuitable for testing programs in the presence of a large number of threads

Bandera [18] is a tool set for model checking concurrent Java software. Given Java source code, it uses slicing to eliminate irrelevant components, abstract interpretation to support data abstraction, and a model-generator to construct finite-state models. Bandera has its own specification language based on temporal specification patterns, which attempts to help users specifying properties to be checked. The models are represented in an intermediate language, which is the input language of Bogor [52]—an extensible framework on which custom-made model checkers can be built. Bogor tries to fill the gap between software semantics and input languages of existing model checkers by supporting modern language features such as polymorphism and virtual methods. Translators from the intermediate language to several model checkers, including Spin, are also available. In contrast to Bandera, jMoped does not check a program against a temporal property, but only performs reachability analyses. jMoped enables users to find out coverability and errors such as assertion violations of programs without an extra effort. LTL model checking, which is supported by Moped, is not built into jMoped.

Unlike the previous model checkers where only finite numbers of states are involved, Microsoft's Slam [3] deals with infinite-state space. The Slam project has been successful in finding bugs in device drivers, written in C language. It implements a technique called *counterexample-guided abstraction refinement* (CEGAR). In this paradigm, a program is first abstracted into a coarse abstraction, which tracks only a few *predicates*—relations between program variables. Then, the model checker Bebop is used to verify safety properties. Because of the nature of the overapproximation, if the verification succeeds, it is guaranteed that the concrete program does not violate the properties. On the other hand, if the verification fails, the error trace can be checked for its feasibility, i.e. whether it corresponds to a concrete program execution or merely introduced by the abstraction. If the trace corresponds to a concrete execution, then a bug has been uncovered. Otherwise, the infeasibility of the error trace is used to refine the abstraction by adding more relevant predicates. The entire process repeats until a bug is found or no new error trace can be found. The process is not guaranteed to terminate, but has been proven to be useful in practice.

The basic CEGAR described above was improved by the model checker Blast [5]. Basically, when an infeasible trace is found, Blast refines the current abstraction by using an interpolation-based algorithm, in which predicates are discovered locally and independently at each program point as interpolants between the past and the future fragments of the trace. Moreover, it implements a so-called lazy abstraction, where the refinement procedure takes place locally, i.e. only in the parts where infeasible trace occurred. CEGAR was also implemented in Moped [22], but is currently not incorporated into jMoped.

Chapter 2

Preliminaries

This chapter contains definitions used throughout the thesis. We introduce three different computational models that operate on stacks: pushdown systems, alternating pushdown systems, and pushdown networks. We also introduce semirings which are used as *weights* in all three pushdown models. At the end of the chapter, we briefly discuss binary decision diagrams, an important data structure that is used for representing weights.

2.1 Basic definitions

Throughout the thesis we denote by $\mathbb{N}$ the set of nonnegative integers and by $\mathbb{R}$ the set of real numbers. Also, $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$, $\mathbb{R}_+ = \{a \in \mathbb{R} \mid a \geq 0\}$, and $\mathbb{R}_+^\infty = \mathbb{R}_+ \cup \{\infty\}$. If n is a positive integer, $[n] = \{a \in \mathbb{N} \mid 1 \leq a \leq n\}$.

Let Σ be an alphabet. A *word* over Σ is a finite sequence of elements of Σ. The *length* of a word w is the number of elements in w, denoted by $|w|$. The empty word is denoted by ε. Σ^* is the set of all words over Σ, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of non-empty words.

Moreover, we sometimes abuse notation by writing singletons without enclosing braces, e.g. p instead of $\{p\}$.

2.1.1 Semirings

A *semiring* is a quintuple $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$, where D is a set, $0, 1 \in D$, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that

1. $(D, \oplus, 0)$ is a commutative monoid with identity element 0.

2. $(D, \otimes, 1)$ is a monoid with identity element 1.

3. 0 is an annihilator with respect to $\otimes$, i.e. for all $a \in D$,

$$a \otimes 0 = 0 \otimes a = 0 \ .$$

4. $\otimes$ distributes over $\oplus$, i.e. for all $a, b, c \in D$,

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \ .$$

A semiring is *idempotent* if $\oplus$ is idempotent, i.e. for all $a \in D$, $a \oplus a = a$. We define a binary relation $\sqsubseteq$ on a semiring: for all $a, b \in D$, $a \sqsubseteq b$ iff $\exists c \in D : a \oplus c = b$. A semiring is *bounded* if there are no infinite ascending chains in the relation $\sqsubseteq$.

Moreover, a semiring is called *naturally ordered* if $\sqsubseteq$ is a partial order. A semiring is *complete* if it is possible to define infinite sums as an extension of finite sums that are associative, commutative, and distributive with respect to $\otimes$. A semiring is ω-continuous if it is naturally ordered, complete, and for all sequences $(a_i)_{i \in \mathbb{N}}$ with $a_i \in D$, $\sup\{\bigoplus_{i=0}^{n} a_i \mid n \in \mathbb{N}\} = \bigoplus_{i \in \mathbb{N}} a_i$.

Let us consider some examples of semirings. The typical ones are the integer semiring $(\mathbb{N}^\infty, +, \times, 0, 1)$ and the real semiring $(\mathbb{R}_+^\infty, +, \times, 0, 1)$. Both are neither bounded nor idempotent. They are ω-continuous with $\sqsubseteq$ as their natural orders in $\mathbb{N}^\infty$ and $\mathbb{R}_+^\infty$, respectively. On the other hand, the tropical semiring $(\mathbb{N}^\infty, \min, +, \infty, 0)$ is bounded, idempotent, and ω-continuous.

Given an alphabet Σ, the language semiring over Σ is $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$, where the operation $\cdot$ is the language concatenation. Given a set S, the binary relation semiring over S is $(2^{S \times S}, \cup, \circ, \emptyset, \{(s, s) \mid s \in S\})$, where the operation $\circ$ is the relation composition. Both semirings are bounded and idempotent. With set inclusion as $\sqsubseteq$, they are also ω-continuous.

We extend the notion of semiring to structures with two extend operators. An *extended semiring* is a tuple $(D, \oplus, \otimes, \odot, 0, 1, 1')$, where

1. $(D, \oplus, 0)$ is a commutative monoid with identity element 0.

2. $(D, \otimes, 1)$ is a monoid with identity element 1.

3. $(D, \odot, 1')$ is a *commutative* monoid with identity element $1'$.

4. 0 is an annihilator with respect to $\otimes$ and $\odot$.

5. $\otimes$ distributes over $\oplus$ and $\odot$.

The notions of idempotence, boundedness, completeness, natural ordering, and ω-continuity are straightforwardly generalized to extended semirings. Note that the notion of completeness is generalized by considering distributivity of infinite sums with respect to both $\otimes$ and $\odot$.

As an example of (bounded idempotent) extended semiring, consider the extended language semiring $(2^{\Sigma^*}, \cup, \cdot, \cap, \emptyset, \{\varepsilon\}, \Sigma^*)$. This structure defines the language accepted by an alternating automaton (see later), where alternation corresponds to the operation of language intersection.

2.1.2 Finite automata

An *alternating automaton* is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where

- Q is a finite set of *states*,
- Σ is a finite *input alphabet*,
- $\delta \subseteq Q \times \Sigma \times 2^Q$ is a set of *transitions*,
- $I \subseteq Q$ is a set of *initial states*, and
- $F \subseteq Q$ is a set of *final states.*

An *automaton* is an alternating automaton where all right-hand sides of transitions are singletons. In that case, we write a transition (q, a, q') instead of $(q, a, \{q'\})$, i.e. braces are dropped.

A *weighted alternating automaton* is an alternating automaton, in which each transition is equipped with an extended semiring value; formally it is a triple $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l)$, where $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is an alternating automaton, $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$ is an extended semiring, and $l : \delta \to D$ is a function that assigns a value from D to each transition in δ. If $l(t) = a$, we say that the transition t has weight a.

Intuitively, a transition $(q, \gamma, \{q'_1, \ldots, q'_n\})$ with weight a of an alternating automaton says that, if the system is at state q receiving the input γ, then the computation of the system forks into n parallel computations, each at state q'_i, for all $i \in [n]$. Therefore, a run can be seen as a tree of computations. The trace of a run is computed from the weights corresponding to the transitions by applying $\otimes$ between successive weights and $\odot$ on the parallel ones.

Depending on directions when computing successive weights, we consider two transition relations $\rightarrow_f, \rightarrow_b \subseteq Q \times \Sigma^* \times D \times 2^Q$, defined as the smallest relations satisfying

- $q \xrightarrow{\varepsilon(1)}_f \{q\}$ and $q \xrightarrow{\varepsilon(1)}_b \{q\}$ for all $q \in Q$, and
- if $t = (q, \gamma, \{q_1, \ldots, q_n\}) \in \delta$, $l(t) = a$, and

  $$q_i \xrightarrow{w(b_i)}_f Q_i \quad \text{resp.} \quad q_i \xrightarrow{w(b_i)}_b Q_i$$

  for each $i \in [n]$, then

  $$q \xrightarrow{\gamma w(a \otimes \bigodot_{i=1}^n b_i)}_f \bigcup_{i=1}^n Q_i \quad \text{resp.} \quad q \xrightarrow{\gamma w(\bigodot_{i=1}^n b_i \otimes a)}_b \bigcup_{i=1}^n Q_i.$$

Notice that in the case of a (non-alternating) automaton, a non-extended semiring is sufficient, and $\rightarrow_f$ and $\rightarrow_b$ always relate states to singletons of states. The second point of the definition above becomes in this case (braces omitted): for a given transition $t = (q, \gamma, q')$ where $l(t) = a$, if $q' \xrightarrow{w(b)}_f q''$ (resp. $q' \xrightarrow{w(b)}_b q''$) then $q \xrightarrow{\gamma w(a \otimes b)}_f q''$ (resp. $q \xrightarrow{\gamma w(b \otimes a)}_b q''$).

Given an initial state q and a word w, we define

$$\begin{aligned} F_{\mathcal{WA}}(q, w) &= \bigoplus\{a \in D \mid \exists Q' \subseteq F : q \xrightarrow{w(a)}_f Q'\} \\ B_{\mathcal{WA}}(q, w) &= \bigoplus\{a \in D \mid \exists Q' \subseteq F : q \xrightarrow{w(a)}_b Q'\} \end{aligned}$$

to be the *forward* and *backward* weight, respectively, of the word w when starting from the state q.

2.2 Pushdown models

We introduce in this section three different computational models that operate on stacks. A stack is a word over some finite *stack alphabet*, which can have unbounded length. We use the term *pushdown model* as a general term to refer to any of the following three models: pushdown systems, alternating pushdown systems, and pushdown networks.

2.2.1 Pushdown systems

A *pushdown system* is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where

- P is a finite set of *control locations*,
- Γ is a finite stack alphabet, and
- $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a set of *transition rules.*

If $((p, \gamma), (p', w)) \in \Delta$, we write $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ instead, and when $|w|$ is 0, 1, and 2, we call the rule *pop*, *normal*, and *push* rule, respectively. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ is a control location and $w \in \Gamma^*$ is a *stack content.*

A *weighted pushdown system* is a pushdown system, in which each rule is equipped with a semiring value; formally $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is a semiring, and $f : \Delta \to D$ is a function that assigns a value from D to each rule in Δ. If $f(\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle) = a$, we often write $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$, or simply append a in parentheses to the rule, i.e. $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ (a), and say that the rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ has *weight a.* We sometimes use the term pushdown system to refer to its weighted version when it is clear from the context.

Intuitively, a transition rule $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$ of a weighted pushdown system says that, if the system is at control location p, in which the top element of the stack is γ, then the system can move to the control state p' and replaces the top symbol γ by the sequence of symbols w (i.e., the operation pops γ and pushes w). A run of a pushdown system is a sequence of successive configurations leading from a configuration to another one. The trace of a run is computed by applying the operation $\otimes$ on all weights corresponding to the transition rules used along the run.

Formally, we define the reachability relation $\Rightarrow \subseteq (P \times \Gamma^*) \times D \times (P \times \Gamma^*)$ to be the smallest relation such that

- $c \overset{1}{\Rightarrow} c$, for all $c \in P \times \Gamma^*$,
- if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$ and $\langle p', ww' \rangle \overset{b}{\Rightarrow} c$ for some $w' \in \Gamma^*$, $b \in D$, and $c \in P \times \Gamma^*$, then $\langle p, \gamma w' \rangle \overset{a \otimes b}{\Longrightarrow} c$.

Given two configurations c and c', we define:

$$T(c, c') = \bigoplus \{a \in D \mid c \overset{a}{\Rightarrow} c'\}$$

to be the traces of all runs starting from c and reaching c'. If $T(c, c') \neq 0$, we say that c' is *reachable* from c.

Consider as an example the weighted pushdown system $\mathcal{WP} = (\mathcal{P}, \mathcal{S}_D, f)$, where $\mathcal{P} = (\{p_0, p_1\}, \{a, b\}, \Delta)$ and $\mathcal{S}_D = (\mathbb{N}^\infty, \min, +, \infty, 0)$—the tropical semiring. The set of transition rules Δ and the function f are defined as follows:

$$\Delta \;=\; \{\langle p_0, a\rangle \overset{1}{\hookrightarrow} \langle p_0, \varepsilon\rangle, \langle p_0, a\rangle \overset{2}{\hookrightarrow} \langle p_1, a\rangle, \langle p_1, a\rangle \overset{3}{\hookrightarrow} \langle p_0, ab\rangle\}\ .$$

Given two configurations $\langle p_0, a\rangle$ and $\langle p_0, b\rangle$, the configuration $\langle p_0, b\rangle$ is reachable from $\langle p_0, a\rangle$ because

$$\langle p_0, a\rangle \overset{2}{\Rightarrow} \langle p_1, a\rangle \overset{3}{\Rightarrow} \langle p_0, ab\rangle \overset{1}{\Rightarrow} \langle p_0, b\rangle\ ,$$

therefore $\langle p_0, a\rangle \xRightarrow{2+3+1} \langle p_0, b\rangle$. Since it is the only possible run from $\langle p_0, a\rangle$ to $\langle p_0, b\rangle$, we have $T(\langle p_0, a\rangle, \langle p_0, b\rangle) = 6$.

2.2.2 Alternating pushdown systems

An *alternating pushdown system* is a generalization of a pushdown system, where right-hand sides of rules are sets of configurations. Formally, an alternating pushdown system is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where

- P is a finite set of control locations,
- Γ is a finite stack alphabet, and
- $\Delta \subseteq (P \times \Gamma) \times 2^{P \times \Gamma^*}$ is a set of *transition rules*.

If $((p, \gamma), \{(p_1, w_1), \dots, (p_n, w_n)\}) \in \Delta$, we write

$$\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, w_n\rangle, \dots, \langle p_n, w_n\rangle\}$$

instead. We call a rule *alternating* if $n > 1$, and *non-alternating* otherwise. We also write $\langle p, \gamma\rangle \hookrightarrow \langle p_1, w_1\rangle$ (braces omitted) for a non-alternating rule. Notice that an alternating pushdown system is a pushdown system if all rules are non-alternating. For a better distinction, we will denote a configuration by a lowercase letter (e.g. c) and a set of configurations by an uppercase letter (e.g. C).

The notion of weight is also generalized. A *weighted alternating pushdown system* is an alternating pushdown system, in which each rule is equipped with an *extended* semiring value; formally, $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is an alternating pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$ is an extended semiring, and $f : \Delta \to D$ is a function that assigns a value from D to each rule in Δ. We sometimes use the term alternating pushdown system to refer to its weighted version when it is clear from the context.

Intuitively, a rule $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\}$ says that, from a configuration c where p is the control location and γ is the top of stack symbol, the computation of the system forks into n parallel computations, each of them starting from the configuration obtained from c by replacing p by p_i and γ by w_i, for all $i \in [n]$. Therefore, a run can be seen as a tree of computations. The trace of a run is computed from the weights corresponding to the transition rules by applying $\otimes$ between successive weights and $\odot$ on the parallel ones.

Formally, we define the reachability relation $\Rightarrow \subseteq (P \times \Gamma^*) \times D \times 2^{P\times\Gamma^*}$ to be the smallest relation such that

- $c \overset{1}{\Rightarrow} \{c\}$, for all $c \in P \times \Gamma^*$,
- if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\}$ and $\langle p_i, w_i w\rangle \overset{b_i}{\Rightarrow} C_i$ for some $w \in \Gamma^*$, $b_i \in D$, and $C_i \subseteq P \times \Gamma^*$, for each $i \in [n]$, then

$$\langle p, \gamma w\rangle \xRightarrow{a\otimes\bigodot_{i=1}^{n} b_i} \bigcup_{i=1}^{n} C_i \ .$$

Given a configuration c and a set of configurations C, we define

$$T(c, C) = \bigoplus\{a \in D \mid c \overset{a}{\Rightarrow} C\}$$

to be the traces of all runs starting from c and reaching (precisely) the set C. If $T(c, C) \neq 0$, we say that c is *backwards reachable* from C.

Consider as an example the weighted alternating pushdown system $\mathcal{WP} = (\mathcal{P}, \mathcal{S}_D, f)$, where $\mathcal{P} = (\{p_0, p_1\}, \{a, b\}, \Delta)$ and $\mathcal{S}_D = (\mathbb{N}^\infty, \min, +, +, \infty, 0, 0)$. The set of transition rules Δ and the function f are defined as follows:

$$\begin{aligned}\Delta \;=\; \{&\langle p_0, a\rangle \overset{1}{\hookrightarrow} \langle p_1, bb\rangle, \langle p_1, b\rangle \overset{2}{\hookrightarrow} \{\langle p_1, a\rangle, \langle p_0, b\rangle\},\\ &\langle p_1, a\rangle \overset{3}{\hookrightarrow} \langle p_0, \varepsilon\rangle, \langle p_0, b\rangle \overset{4}{\hookrightarrow} \langle p_0, \varepsilon\rangle\} \ .\end{aligned}$$

Given a configuration $\langle p_0, a\rangle$ and a set of configurations $\{\langle p_0, b\rangle\}$, the configuration $\langle p_0, a\rangle$ is backwards reachable from $\{\langle p_0, b\rangle\}$ because

$$\begin{aligned}
\langle p_0, a\rangle &\overset{1}{\Rightarrow} \{\langle p_1, bb\rangle\}\ ,\\
\langle p_1, bb\rangle &\overset{2}{\Rightarrow} \{\langle p_1, ab\rangle, \langle p_0, bb\rangle\}\ ,\\
\langle p_1, ab\rangle &\overset{3}{\Rightarrow} \{\langle p_0, b\rangle\}\ ,\\
\langle p_0, bb\rangle &\overset{4}{\Rightarrow} \{\langle p_0, b\rangle\}\ ,
\end{aligned}$$

so $\langle p_0, a\rangle \xRightarrow{1+2+(3+4)} \{\langle p_0, b\rangle\}$. Since it is the only possible run from $\langle p_0, a\rangle$ to $\{\langle p_0, b\rangle\}$, we have $T(\langle p_0, a\rangle, \{\langle p_0, b\rangle\}) = 10$.

2.2.3 Pushdown networks

A *pushdown network* is a set of pushdown systems, each of them represents a *process*. Control locations and stack symbols are shared among processes. However, for a reason that will become clear later, control locations will be called *globals* in the context of pushdown networks. Formally, a pushdown network of n process is a triple $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i\in[n]})$, where

- G is a finite set of globals,
- Γ is a finite stack alphabet, and
- $\Delta_i \subseteq (G \times \Gamma) \times (G \times \Gamma^*)$, for each $i \in [n]$, is a set of transition rules for the i-th process.

In contrast to a global, a *local* is a word over the stack alphabet. A *local configuration* of $\mathcal{N}$ is a configuration of a pushdown system, i.e. a pair of a global and a local $\langle g, w\rangle \in G \times \Gamma^*$. A *global configuration* of $\mathcal{N}$ is a tuple of a global and n locals $\langle g, w_1, \ldots, w_n\rangle \in G \times (\Gamma^*)^n$. We will denote local configurations by lowercase letters (e.g. c) and global configurations by uppercase letters (e.g. C). Intuitively, the network consists of n processes, each of which have some local storage (i.e., the local storage of the i-th process is the word w_i), and the processes can communicate by reading and manipulating the global storage represented by g. Notice that a pushdown network is a pushdown system when $n = 1$.

Similarly, a *weighted pushdown network* is a triple $\mathcal{WN} = (\mathcal{N}, \mathcal{S}, (f_i)_{i\in[n]})$, where $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i\in[n]})$ is a pushdown network, $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is a semiring, and for $i \in [n]$, $f_i : \Delta_i \to D$ is a function that assigns a value from D_i to each rule in Δ_i. We sometimes use the term pushdown network to refer to its weighted version when it is clear from the context.

For each $i \in [n]$, the local reachability relation $\Rightarrow_i$ is defined as for a pushdown system. The global reachability relation $\Rightarrow \subseteq G \times (\Gamma^*)^n \times D \times G \times (\Gamma^*)^n$ is defined as follows:

$$\begin{gathered}\langle g, w_1, \ldots, w_i, \ldots, w_n \rangle \overset{a}{\Rightarrow} \langle g', w_1, \ldots, w_i', \ldots, w_n \rangle \\ \text{iff } \exists i \in [n] : \langle g, w_i \rangle \overset{a}{\Rightarrow}_i \langle g', w_i' \rangle \ .\end{gathered}$$

Given two global configurations C and C', we define

$$T(C, C') = \bigoplus \{a \in D \mid C \overset{a}{\Rightarrow} C'\}$$

to be the traces of all runs starting from C and reaching C'. If $T(C, C') \neq 0$, we say that C' is *reachable* from C.

Consider as an example the weighted pushdown network $(\mathcal{N}, \mathcal{S}_D, (f_1, f_2))$, where $\mathcal{N} = (\{g_0, g_1\}, \{a, b\}, (\Delta_1, \Delta_2))$ and $\mathcal{S}_D = (\mathbb{N}^\infty, \min, +, \infty, 0)$. The sets of transition rules Δ_1, Δ_2 and the functions f_1, f_2 are defined as follows:

$$\begin{aligned}\Delta_1 &= \{\langle g_0, a \rangle \overset{1}{\hookrightarrow} \langle g_1, aa \rangle\} \ , \\ \Delta_2 &= \{\langle g_1, b \rangle \overset{2}{\hookrightarrow} \langle g_0, bb \rangle\} \ .\end{aligned}$$

Given two global configurations $\langle g_0, a, b \rangle$ and $\langle g_1, aaa, bb \rangle$, the configuration $\langle g_1, aaa, bb \rangle$ is reachable from $\langle g_0, a, b \rangle$ because

$$\langle g_0, a, b \rangle \overset{1}{\Rightarrow} \langle g_1, aa, b \rangle \overset{2}{\Rightarrow} \langle g_0, aa, bb \rangle \overset{1}{\Rightarrow} \langle g_1, aaa, bb \rangle \ ,$$

therefore $\langle g_0, a, b \rangle \xRightarrow{1+2+1} \langle g_1, aaa, bb \rangle$. Since it is the only possible run from $\langle g_0, a, a \rangle$ to $\langle g_1, aaa, bb \rangle$, we have $T(\langle g_0, a, a \rangle, \langle g_1, aaa, bb \rangle) = 4$.

2.3 Binary decision diagrams

We briefly discuss in this section an important data structure that is used later in the thesis for representing weights in pushdown models as relations over a finite domain. We proceed by following the style of [53]. Let n be a positive integer. The aim is to find a data structure that is able to compactly represent subsets of $\{0,1\}^n$ in hopes that the representations are smaller than explicit enumerations of the subsets.

One possibility is to represent the subsets as Boolean formulas. Given a set of Boolean variables $V = \{v_1, \ldots, v_n\}$, Boolean formulas t are defined by the following grammar:

$$t ::= 0 \mid 1 \mid v_i \mid \neg t \mid t \vee t \ ,$$

where $v_i \in V$, and 0 and 1 denote the constants false and true, respectively. With negation ($\neg$) and disjunction ($\vee$) having their usual meanings, given a valuation $\mathcal{V} : V \to \{0,1\}$ and a Boolean formula t, it is possible to evaluate whether $\mathcal{V}$ satisfies t, written $\mathcal{V} \models t$. Therefore, we can always associate the formula t with the set $\{\mathcal{V} \mid \mathcal{V} \models t\}$, i.e. the set of valuations that satisfy t. This means also that we can represent a set by its corresponding Boolean formula. Standard set operations, e.g. union, become Boolean operations, e.g. disjunction. Indeed, in later chapters we freely interchange sets and Boolean formulas as well as their operations.

A binary decision diagram (BDD) is a data structure that is used to represent a Boolean formula. A BDD with domain V is a rooted, directed, acyclic graph, which consists of nodes labeled by elements of V and two terminal nodes labeled by 0 and 1. The terminal nodes have no outgoing edges. Each non-terminal node has two child nodes, with outgoing edges labeled by 0 and 1.

A path from the root node to the terminal node 1 represents a variable valuation $\mathcal{V}$ that satisfies the represented Boolean formula. The valuation of a variable v is 0 (resp. 1) if the path traverses the node labeled by v with the edge labeled by 0 (resp. 1). So, a BDD represents a Boolean formula t in the following manner: $\mathcal{V} \models t$ if and only if there is a path in the BDD that represents $\mathcal{V}$. Figure 2.1 gives an example of a BDD with domain $\{x, y, x', y'\}$. The BDD represents the formula $x' \wedge (y' \leftrightarrow x \vee y)$. Clearly, the BDD represents the set $\{(0,0,1,0), (0,1,1,1), (1,0,1,1), (1,1,1,1)\}$, where quadruples represent valuations of x, y, x', and y', respectively,

A BDD is *ordered* if there exists a total order $<$ on V such that different variables appear in the same order on every path from the root. We write $v < v'$ if there is an edge from a node labeled by v to a node labeled by v'. The BDD in the example has the ordering $x < y < x' < y'$. A BDD is *reduced* if it does not contain (i) isomorphic subgraphs, and (ii) nodes whose children are the same. Figure 2.2 shows the BDD reduced from the one in

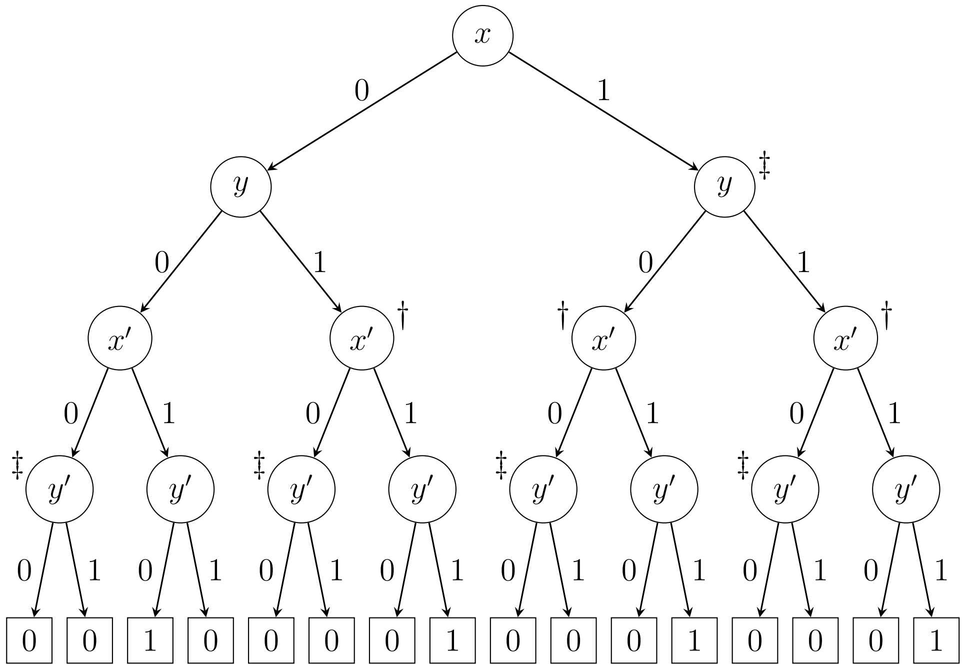


Figure 2.1: A BDD for the formula $x' \wedge (y' \leftrightarrow x \vee y)$.

Figure 2.1. All terminal nodes are merged into only two leaves. Subgraphs marked with † in Figure 2.1 are reduced due to isomorphism. The nodes labeled with ‡ are eliminated as both children are the same.

In a reduced ordered BDD, a path from the root node to the terminal node 1 can represent more than one valuation when the number of nodes on the path is less than the number of variables. The idea is that the valuations of variables specified by the path always satisfy the underlying Boolean formula, independent of the valuations of variables of the missing nodes. For instance, the rightmost path in Figure 2.2 skips the node for y, thus representing two valuations, namely $(1, 0, 1, 1)$ and $(1, 1, 1, 1)$.

An important property of reduced ordered BDDs is their canonicity, i.e. the BDD for a given Boolean formula (and a given order $<$) is always unique. This property makes it possible to check whether two BDDs are equivalent in constant time. Further treatments on BDDs are beyond the scope of the thesis. The reader is referred to [2] for detailed discussions on the topic. Nevertheless, since we only deal with reduced ordered BDDs throughout the thesis, we drop the prefix from now on, and merely use the term BDD to refer to its reduced ordered version.

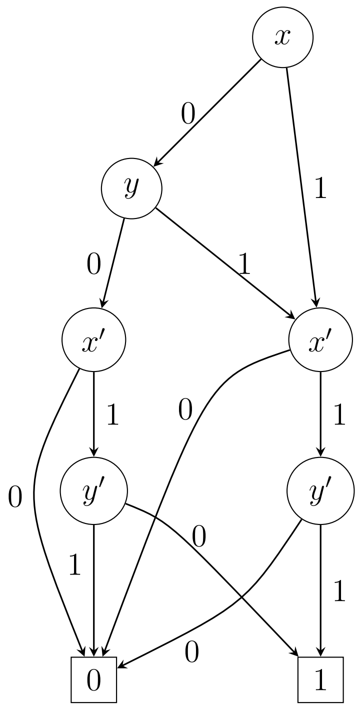


Figure 2.2: The BDD for $x' \wedge (y' \leftrightarrow x \vee y)$, reduced from Figure 2.1.

Chapter 3

Reachability analyses

In this chapter we study reachability analyses for the pushdown models introduced in Chapter 2. We divide the study into two sections based on the types of weights, namely bounded idempotent semirings and the general ones.

3.1 Bounded idempotent semirings

We propose in this section reachability algorithms for each of our three pushdown models when weights are taken from bounded idempotent semirings. The algorithms have the same flavor, in the sense that they all use similar data structures for representing possibly infinite sets of configurations.

3.1.1 Pushdown systems

Throughout this section, let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ denote a fixed weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$. Since $\mathcal{WP}$ can have infinitely many configurations, we need an appropriate data structure to represent sets of configurations. We use finite automata for this purpose.

A *weighted* $\mathcal{WP}$*-automaton* is a weighted automaton which takes Γ as the input alphabet and P as the initial states; formally $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l)$, where $\mathcal{A} = (Q, \Gamma, \delta, P, F)$. The *weight* of the configuration $\langle p, w \rangle$ is $B_{\mathcal{WA}}(p, w)$—the backward weight of w when starting from the state p (see Section 2.1.2). We say that $\langle p, w \rangle$ is *accepted* by $\mathcal{WA}$ if its weight is non-zero, i.e. $B_{\mathcal{WA}}(p, w) \neq 0$. The set of configurations accepted by $\mathcal{WA}$ is denoted by $L(\mathcal{WA})$. A set of configurations is *regular* if it is accepted by some weighted $\mathcal{WP}$-automaton.

We often drop $\mathcal{WP}$ and say only weighted automaton when $\mathcal{WP}$ is understood.

The left part of Figure 3.1 illustrates a weighted $\mathcal{WP}$-automaton, where $\mathcal{WP}$ is taken from the example in Section 2.2.1. Its rules are listed again on the lower left part of the figure. The figure defines the automaton $\mathcal{WA} = (\mathcal{A}, \mathcal{S}_D, l_0)$, where $\mathcal{A} = (\{p_0, p_1, q_0, q_1\}, \{a, b\}, \delta_0, \{p_0, p_1\}, \{q_1\})$. Each transition in the figure is labeled with an input symbol followed by a weight in parentheses, e.g. $p_0 \xrightarrow{a(2)} q_0$ means $(p_0, a, q_0) \in \delta_0$ and $l_0(p_0, a, q_0) = 2$. One can see that $\mathcal{WA}$ accepts the configuration $\langle p_0, ab(bb)^n\rangle$, for all $n \geq 0$, with weight $B_{\mathcal{WA}}(p_0, ab(bb)^n) = 3 + 2n$.

Besides $\mathcal{WP}$, let us fix a weighted automaton $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l_0)$, where $\mathcal{A} = (Q_0, \Gamma, \delta_0, P, F)$, such that $L(\mathcal{WA}) = C$ for some $C \subseteq P \times \Gamma^*$. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We propose in this section an algorithm that constructs $\mathcal{WA}_{post^*}$ accepting the set of configurations that are reachable from any configuration $c \in C$ with weights computed by applying the operation $\otimes$ on the weight of c and the traces starting from c. Formally, $\mathcal{WA}_{post^*}$ accepts

$$post^*(C) = \{c' \in P \times \Gamma^* \mid \exists c \in C : T(c, c') \neq 0\}$$

such that the weight of a configuration $c' \in post^*(C)$ is defined as

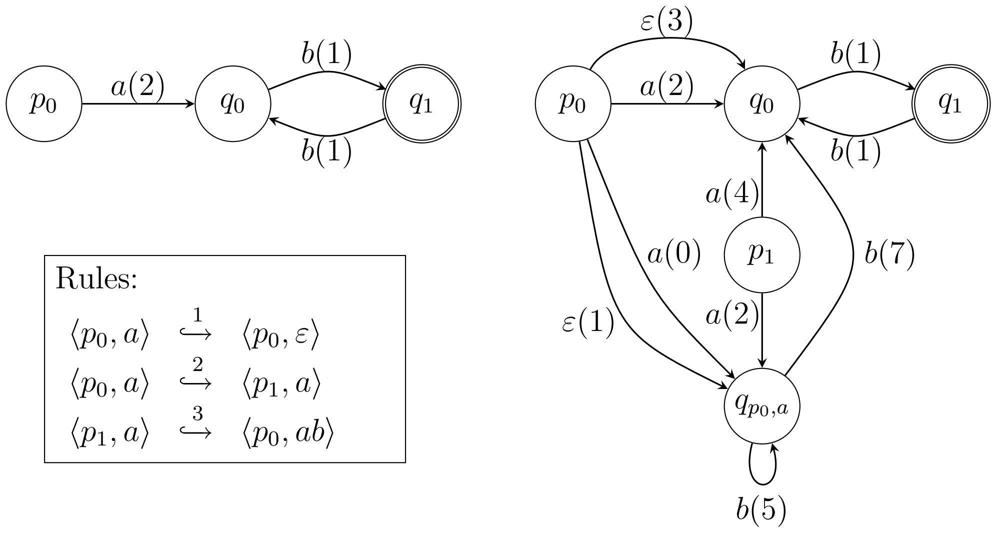
$$B_{\mathcal{WA}_{post^*}}(c') = \bigoplus_{c \in C} B_{\mathcal{WA}}(c) \otimes T(c, c') \ .$$

Without loss of generality we assume from now on that for every rule $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, we have $|w| \leq 2$. Note that this does not restrict the expressiveness, since every pushdown system can be simulated by one that satisfies this restriction with only a linear increase in size. For instance, the rule $\langle p, a\rangle \overset{x}{\hookrightarrow} \langle p', bcde\rangle$ can be simulated by introducing fresh stack symbols f, g and the rules

$$\langle p, a\rangle \overset{x}{\hookrightarrow} \langle p', fe\rangle,\ \langle p', f\rangle \overset{1}{\hookrightarrow} \langle p', gd\rangle,\ \langle p', g\rangle \overset{1}{\hookrightarrow} \langle p', bc\rangle \ .$$

Saturation procedure

A saturation procedure for constructing non-weighted $\mathcal{WA}_{post^*}$ was first proposed in [21]. The following generalizes it to the the weighted case.



Rules:

$$\begin{array}{lcl} \langle p_0, a\rangle & \overset{1}{\hookrightarrow} & \langle p_0, \varepsilon\rangle \\ \langle p_0, a\rangle & \overset{2}{\hookrightarrow} & \langle p_1, a\rangle \\ \langle p_1, a\rangle & \overset{3}{\hookrightarrow} & \langle p_0, ab\rangle \end{array}$$

Figure 3.1: The weighted automata $\mathcal{WA}$ (left) and $\mathcal{WA}_{post^*}$ (right)

Formally, the weighted automaton $\mathcal{WA}_{post^*}$ is defined as $(\mathcal{A}_{post^*}, \mathcal{S}, l)$, where $\mathcal{A}_{post^*} = (Q, \Gamma, \delta, P, F)$. The set of states of $\mathcal{A}_{post^*}$ is

$$Q = Q_0 \cup \{q_{p',\gamma'} \mid \exists p, \gamma, \gamma'' : \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta\} \ .$$

Initially, $\delta = \delta_0$ and $l(t) = l_0(t)$, if $t \in \delta_0$ and $l(t) = 0$, otherwise. We iteratively update δ and l according to the following saturation rules until no values can be updated, i.e. until the automaton is *saturated*.

1. If $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \varepsilon\rangle$ and $p \xrightarrow{\gamma(b)} q$ in the current automaton, add (p', ε, q) to δ, and update $l(p', \varepsilon, q) = l(p', \varepsilon, q) \oplus (b \otimes a)$.

2. If $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\rangle$ and $p \xrightarrow{\gamma(b)} q$ in the current automaton, add (p', γ', q) to δ, and update $l(p', \gamma', q) = l(p', \gamma', q) \oplus (b \otimes a)$.

3. If $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle$ and $p \xrightarrow{\gamma(b)} q$ in the current automaton, add $(p', \gamma', q_{p',\gamma'})$ and $(q_{p',\gamma'}, \gamma'', q)$ to δ, and update $l(p', \gamma', q_{p',\gamma'}) = 1$ and $l(q_{p',\gamma'}, \gamma'', q) = l(q_{p',\gamma'}, \gamma'', q) \oplus (b \otimes a)$.

Consider again the example in Figure 3.1. The automaton $\mathcal{WA}_{post^*}$ on the right is the output of the saturation procedure when running with $\mathcal{WP}$ and $\mathcal{WA}$ described on the left. We explain some major steps of the procedure in the following.

1. The set of states Q includes the new state $q_{p_0,a}$ because of the rule $\langle p_1, a\rangle \hookrightarrow \langle p_0, ab\rangle$.

2. $\mathcal{WA}_{post^*}$ is initialized with the transitions and weights from $\mathcal{WA}$, i.e. the transitions $(p_0, a, q_0), (q_0, b, q_1), (q_1, b, q_0)$ together with their corresponding weights are added to $\mathcal{WA}_{post^*}$.

3. The transition (p_0, a, q_0) matches the rule

   - $\langle p_0, a\rangle \overset{1}{\hookrightarrow} \langle p_0, \varepsilon\rangle$, so the transition (p_0, ε, q_0) with weight $2+1 = 3$ is added, and
   - $\langle p_0, a\rangle \overset{2}{\hookrightarrow} \langle p_1, a\rangle$, so the transition (p_1, a, q_0) with weight $2+2 = 4$ is added.

4. The transition (p_1, a, q_0) matches the rule $\langle p_1, a\rangle \overset{3}{\hookrightarrow} \langle p_0, ab\rangle$, so the transitions $(p_0, a, q_{p_0,a})$ with weight 0 and $(q_{p_0,a}, b, q_0)$ with weight $4 + 3 = 7$ are added.

5. We now consider $(p_0, a, q_{p_0,a})$, and continue analogously to the steps 3–4; essentially the following transitions are added: $(p_0, \varepsilon, q_{p_0,a})$ with weight 1, $(p_1, a, q_{p_0,a})$ with weight 2, $(q_{p_0,a}, b, q_{p_0,a})$ with weight 5. Note that $(p_0, a, q_{p_0,a})$ is not added again because its weight does not change.

6. No new transitions can be updated, i.e. $\mathcal{WA}_{post^*}$ is saturated.

As an example, the weight of the configuration $\langle p_0, abbb\rangle$ accepted by $\mathcal{WA}_{post^*}$ is

$$B_{\mathcal{WA}_{post^*}}(p_0, abbb) = \min(1 + 1 + 1 + 2, 1 + 7 + 5 + 0) = 5 \ .$$

Implementation

Algorithm 3.1, proposed by [51], efficiently implements the saturation procedure. Lines 1–3 initialize the algorithm as described above. Initially, *trans* is equal to δ_0. The algorithm then repeatedly removes a transition (p, γ, q) from *trans* until it is empty. The loops at lines 7, 9, and 11 handle the cases when p and γ match left-hand sides of the rules; resembling the saturation rules 1, 2, and 3, respectively. The loops at lines 14 and 17 handle ε-transitions. New transitions and weights are produced as a consequence.

Input: Weighted pushdown system $(\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$,
and weighted automaton $(\mathcal{A}, \mathcal{S}, l_0)$, where $\mathcal{A} = (Q_0, \Gamma, \delta_0, P, F)$
Output: The saturated weighted automaton $\mathcal{WA}_{post^*}$

```
$Q := Q_0 \cup \{q_{p',\gamma'} \mid \exists p, \gamma, \gamma'' : \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta\}$;
$\delta := \delta_0$; $trans := \delta_0$; $l := \lambda t.0$;
forall $t \in \delta_0$ do $l(t) := l_0(t)$;
while $trans \neq \emptyset$ do
    remove $t := (p, \gamma, q)$ from $trans$;
    if $\gamma \neq \varepsilon$ then
        forall $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ do
            update($(p', \varepsilon, q)$, $l(t) \otimes f(r)$);
        forall $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ do
            update($(p', \gamma', q)$, $l(t) \otimes f(r)$);
        forall $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ do
            update($(p', \gamma', q_{p',\gamma'})$, $1$);
            update($(q_{p',\gamma'}, \gamma'', q)$, $l(t) \otimes f(r)$);
            forall $t' = (p'', \varepsilon, q_{p',\gamma'}) \in \delta$ do
                update($(p'', \gamma'', q_{p',\gamma'})$, $l(t) \otimes f(r) \otimes l(t')$);
    else
        forall $t' = (q, \gamma', q') \in \delta$ do
            update($(p, \gamma', q')$, $l(t') \otimes l(t)$);
    return $((Q, \Gamma, \delta, P, F), \mathcal{S}, l)$;
```

Algorithm 3.1: A reachability algorithm for weighted pushdown systems

The procedure `update` listed in Algorithm 3.2 is called when a new weight of a transition is computed. The new transition is added to δ before combining the new weight with the old weight (lines 2–3). The if-statement at line 4 makes sure that the transition is added to *trans* for further computation (line 5) only if its weight changes. As a result, line 6 can change $l(t)$ only to a larger value (with respect to $\sqsubseteq$). The algorithm continues until the weights of all transitions stabilize, i.e. *trans* is empty.

```
1 procedure update($t, v$)
2     $\delta := \delta \cup \{t\}$;
3     $v := l(t) \oplus v$;
4     if $v \neq l(t)$ then
5         $trans := trans \cup \{t\}$;
6         $l(t) := v$;
```

Algorithm 3.2: The update procedure

If Algorithm 3.1 is applied to the example in Figure 3.1, one will obtain an identical result except that ε-transitions are explicitly extended with neighboring transitions (by lines 14 and 17 of the algorithm). The transitions (p_0, b, q_1) with weight 4, $(p_0, b, q_{p_0,a})$ with weight 6, and (p_0, b, q_0) with weight 8 are added as a result.

Theorem 3.1 [51] *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S}$ is a bounded idempotent semiring having the maximal length of ascending chains c. Let $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l)$ be a weighted $\mathcal{WP}$-automaton, where $\mathcal{A} = (Q, \Gamma, \delta, P, F)$. There exists a weighted $\mathcal{WP}$-automaton $\mathcal{WA}_{post^*}$ such that $L(\mathcal{WA}_{post^*}) = post^*(L(\mathcal{WA}))$ and for all $c' \in L(\mathcal{WA}_{post^*})$, $B_{\mathcal{WA}_{post^*}}(c') = \bigoplus_{c \in L(\mathcal{WA})} B_{\mathcal{WA}}(c) \otimes T(c, c')$. Moreover, $\mathcal{WA}_{post^*}$ can be constructed in $O(c|P|(|\Delta|(n_1 + n_2) + |\delta_0|))$ time, where $n_1 = |Q \setminus P|$, and n_2 is the number of different pairs (p', γ') such that there is a rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$.*

3.1.2 Alternating pushdown systems

Notice that the algorithm for pushdown systems in the previous section generates all configurations that are reachable from a given set of configurations. By its characteristics, the algorithm can be categorized as a forward reachability analysis. In contrast, we propose in this section a *backward* reachability analysis for alternating pushdown systems, i.e. given a set of configurations C we find all configurations that are *backwards* reachable from any subsets of C.

We proceed analogously by first introducing a data structure for sets of configurations. Let us fix a weighted alternating pushdown system $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$. A *weighted alternating $\mathcal{WP}$-automaton* is straightforwardly generalized from weighted $\mathcal{WP}$-automaton to the alternating level. The weight of a configuration $\langle p, w\rangle$ is $F_{\mathcal{WA}}(p, w)$—the forward weight of w when starting from the state p (see Section 2.1.2). We say that $\langle p, w\rangle$ is accepted by $\mathcal{WA}$ if $F_{\mathcal{WA}}(p, w) \neq 0$. The set of configurations accepted by $\mathcal{WA}$ is denoted by $L(\mathcal{WA})$.

We fix further a weighted alternating automaton $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l_0)$, where $\mathcal{A} = (Q, \Gamma, \delta_0, P, F)$ such that $L(\mathcal{WA}) = C$ for some $C \subseteq P \times \Gamma^*$. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We present in this section an algorithm that constructs $\mathcal{WA}_{pre^*}$ accepting a set of configurations that are backwards reachable from any subset of configurations $C' \subseteq C$ with weights computed by applying the operation $\otimes$ on the traces reaching C' and $\odot$ on the weights of configurations in C'. Formally, $\mathcal{WA}_{pre^*}$ accepts

$$pre^*(C) = \{c \in P \times \Gamma^* \mid \exists C' \subseteq C : T(c, C') \neq 0\}$$

such that the weight of a configuration $c \in pre^*(C)$ is defined as

$$F_{\mathcal{WA}_{pre^*}}(c) = \bigoplus_{C' \subseteq C} \left(T(c, C') \otimes \bigodot_{c' \in C'} F_{\mathcal{WA}}(c') \right) .$$

Saturation procedure

A saturation procedure for constructing non-weighted $\mathcal{WA}_{pre^*}$ was first introduced in [6]. The following generalizes it to the the weighted case.

The weighted alternating automaton $\mathcal{WA}_{pre^*}$ is defined as $(\mathcal{A}_{pre^*}, \mathcal{S}, l)$, where $\mathcal{A}_{pre^*} = (Q, \Gamma, \delta, P, F)$. Initially, $\delta = \delta_0$ and $l(t) = l_0(t)$, if $t \in \delta_0$ and $l(t) = 0$, otherwise. We iteratively update δ and l according to the following saturation rule until no values can be updated, i.e. until the automaton is *saturated*.

If $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\}$ and $p_i \xrightarrow{w_i(b_i)} Q_i$ for all $i \in [n]$, add $t = (p, \gamma, \bigcup_{i=1}^{n} Q_i)$ to δ and update $l(t) = l(t) \oplus (a \otimes \bigodot_{i=1}^{n} b_i)$.

We reuse the example in Section 2.2.2 to illustrate the idea. Let $\mathcal{WA} = (\mathcal{A}, \mathcal{S}_D, l_0)$ be a weighted alternating $\mathcal{WP}$-automaton, where

$$\mathcal{A} = (\{p_0, p_1, q\}, \{a, b\}, \{(p_0, b, q), (q, b, q)\}, \{p_0, p_1\}, \{q\}) \ ,$$

$l_0(p_0, b, q) = 2$ and $l_0(q, b, q) = 1$. Obviously, $\mathcal{WA}$ accepts of the configuration $\langle p_0, bb^n \rangle$, for all $n \geq 0$, with weight $F_{\mathcal{WA}}(p_0, bb^n) = 2 + n$.

Using the saturation procedure, we now construct the automaton $\mathcal{WA}_{pre^*}$ from $\mathcal{WA}$ and $\mathcal{WP}$. The steps of the procedure are listed in the following.

1. $\mathcal{WA}_{pre^*}$ is initialized with the transitions and weights from $\mathcal{WA}$.

2. Since $\langle p_1, a \rangle \overset{3}{\hookrightarrow} \langle p_0, \varepsilon \rangle$, $\langle p_0, b \rangle \overset{4}{\hookrightarrow} \langle p_0, \varepsilon \rangle$, and $p_0 \xrightarrow{\varepsilon(0)} \{p_0\}$, the transitions (p_1, a, p_0) and (p_0, b, p_0) are added with weights 3 and 4, respectively.

3. With $\langle p_1, b \rangle \overset{2}{\hookrightarrow} \{\langle p_1, a \rangle, \langle p_0, b \rangle\}$ and $p_1 \xrightarrow{a(3)} \{p_0\}$, two transitions are added, because there are two transitions leaving p_0 with symbol b.

    - With $p_0 \xrightarrow{b(4)} \{p_0\}$, the transition (p_1, b, p_0) is added with weight $2 + (3 + 4) = 9$.
    - With $p_0 \xrightarrow{b(2)} \{q\}$, the transition $(p_1, b, \{p_0, q\})$ is added with weight $2 + (3 + 2) = 7$.

4. With $\langle p_0, a \rangle \overset{1}{\hookrightarrow} \langle p_1, bb \rangle$ and $p_1 \xrightarrow{b(9)} \{p_0\}$, two transitions are added, because there are two transitions leaving p_0 with symbol b.

    - With $p_0 \xrightarrow{b(4)} \{p_0\}$, the transition (p_0, a, p_0) is added with weight $1 + (9 + 4) = 14$.
    - With $p_0 \xrightarrow{b(2)} \{q\}$, the transition (p_0, a, q) is added with weight $1 + (9 + 2) = 12$.

5. With $\langle p_0, a \rangle \overset{1}{\hookrightarrow} \langle p_1, bb \rangle$ and $p_1 \xrightarrow{b(7)} \{p_0, q\}$, two transitions are added, because there are two transitions leaving p_0 with symbol b and one transition leaving q with symbol b.

    - With $p_0 \xrightarrow{b(4)} \{p_0\}$ and $q \xrightarrow{b(1)} \{q\}$, the transition $(p_0, a, \{p_0, q\})$ is added with weight $1 + (7 + (4 + 1)) = 13$.

- With $p_0 \xrightarrow{b(2)} \{q\}$ and $q \xrightarrow{b(1)} \{q\}$, the weight of the transition (p_0, a, q) is updated with $\min(12, 1 + (7 + (2 + 1))) = 11$.

6. No new transitions can be added or updated, i.e. $\mathcal{WA}_{pre^*}$ is saturated.

As an example, the weight of the configuration $\langle p_1, bb\rangle$ accepted by $\mathcal{WA}_{pre^*}$ is the combination of the following two traces: (i) $p_1 \xrightarrow{b(9)} \{p_0\}$ and $p_0 \xrightarrow{b(2)} \{q\}$, (ii) $p_1 \xrightarrow{b(7)} \{p_0, q\}$, $p_0 \xrightarrow{b(2)} \{q\}$, and $q \xrightarrow{b(1)} \{q\}$.

$$F_{\mathcal{WA}_{pre^*}}(p_1, bb) = \min(9 + 2, 7 + (2 + 1)) = 10 \ .$$

Implementation

Algorithm 3.3 presents an implementation of the saturation procedure. Without loss of generality, the algorithm imposes two restrictions on every rule $\langle p, \gamma\rangle \hookrightarrow R$ in Δ:

(R1) if $R = \{\langle p', w'\rangle\}$, then $|w'| \leq 2$, and

(R2) if $|R| > 1$, then $\forall \langle p', w'\rangle \in R : |w'| = 1$.

Note that any alternating pushdown system can be converted into an equivalent one that satisfies (R1) and (R2) with only a linear increase in size.

Lines 1–6 initialize the algorithm. Initially, *trans* contains transitions from δ_0 plus an extra *flag* which is false (`ff`) by default. A flag of a transition indicates how the transition was added to *trans*: false means that the transition was added in line 1 or by the procedure `update`, and true (`tt`) means that the transition was added as a result of processing a push rule (see later). Therefore, the procedure `update` used by the algorithm is slightly modified from Algorithm 3.2 by changing line 5 to

```
5 $trans := trans \cup \{(q, \gamma, Q', \mathtt{ff}) \mid t = (q, \gamma, Q')\}$;
```

All rules are copied to Δ' (line 3), and the auxiliary function $\mathcal{F}$ is initialized to map any rule $r \in \Delta$ to a set containing $(f(r), 1', \emptyset)$ as the only element (line 4). Pop rules and rules having the empty set as the right-hand sets are dealt with first (lines 5 and 6). The algorithm then proceeds by iteratively removing transitions from *trans* (line 8), and examining whether they generate other transitions via the saturation rule (lines 9–18). The idea

Input: Weighted alternating pushdown system $(\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, and weighted alternating automaton $(\mathcal{A}, \mathcal{S}, l_0)$, where $\mathcal{A} = (Q, \Gamma, \delta_0, P, F)$

Output: The saturated weighted alternating automaton $\mathcal{WA}_{pre^*}$

```
$\delta := \delta_0$; $trans := \{(q, \gamma, Q', \mathtt{ff}) \mid (q, \gamma, Q') \in \delta_0\}$; $l := \lambda t.0$;
forall $t \in \delta_0$ do $l(t) := l_0(t)$;
$\Delta' := \Delta$; $\mathcal{F} := \lambda r.\emptyset$;
forall $r \in \Delta$ do $\mathcal{F}(r) := \{(f(r), 1', \emptyset)\}$;
forall $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ do update($(p, \gamma, p')$, $f(r)$);
forall $r = \langle p, \gamma \rangle \hookrightarrow \emptyset \in \Delta$ do update($(p, \gamma, \emptyset)$, $f(r)$);
while $trans \neq \emptyset$ do
    remove $(q, \gamma, Q', z)$ from $trans$;
    forall $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in \Delta'$ and $(a, b, Q'') \in \mathcal{F}(r)$ do
        update($(p_1, \gamma_1, Q' \cup Q'')$, $a \otimes (l(q, \gamma, Q') \odot b)$);
    forall $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta'$ and $z = \mathtt{ff}$ do
        add $r' := \langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q', \gamma_2 \rangle \mid q' \in Q'\}$ to $\Delta'$;
        add $(f(r) \otimes l(q, \gamma, Q'), 1', \emptyset)$ to $\mathcal{F}(r')$;
        forall $(q', \gamma_2, Q'') \in \delta$ s.t. $q' \in Q'$ do
            add $(q', \gamma_2, Q'', \mathtt{tt})$ to $trans$;
    forall $r = \langle p, \gamma \rangle \hookrightarrow \{\langle q, \gamma \rangle\} \cup R \in \Delta'$ s.t. $R \neq \emptyset$ do
        add $r' := \langle p, \gamma \rangle \hookrightarrow R$ to $\Delta'$;
        add $\{(a, l(q, \gamma, Q') \odot b, Q' \cup Q'') \mid (a, b, Q'') \in \mathcal{F}(r)\}$ to $\mathcal{F}(r')$;
return $((Q, \Gamma, \delta, P, F), \mathcal{S}, l)$;
```

Algorithm 3.3: A reachability algorithm for weighted alternating pushdown systems

of the algorithm is to avoid unnecessary operations. Imagine that the saturation rule allows to add transition t if transitions t_1 and t_2 are already present. Now, if t_1 is taken from *trans* but t_2 has not been added to $\mathcal{WA}_{pre^*}$, we do not put t_1 back to *trans* but store the following information instead: if t_2 is added, then we can also add t. It turns out that these implications can be stored in the form of the auxiliary sets $\mathcal{F}(r)$.

Let us now look at lines 9–18 in more detail. Line 9 handles normal rules where new transitions can be immediately added. Push rules (lines 11–15) and alternating rules (lines 16–18), however, require a more delicate treatment. At line 11 we know that (q, γ, Q') is a transition of $\mathcal{WA}_{pre^*}$ (because it has been removed from *trans*) and that $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle$ is a push rule of $\mathcal{P}$. We create the "fake rule" $\langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q', \gamma_2 \rangle \mid q' \in Q'\}$, and add it to Δ' at line 12. Its $\mathcal{F}$-value is updated to include the new weight $f(r) \otimes l(q, \gamma, Q')$. Obviously, if Q' contains q' as the only element, then the right-hand set of the fake rule is a singleton, and when a transition (q', γ_2, Q'_2) is later examined (line 9), we update the transition (p_1, γ_1, Q'_2) with weight $f(r) \otimes l(q, \gamma, q') \otimes l(q', \gamma_2, Q''_2)$. On the other hand, if Q' is not a singleton, we know that the resulting fake rule is alternating, and we treat it as a normal alternating rule because it always satisfies the restriction (R2). Also, we need to "reconsider" transitions leaving from a state in Q' with symbol γ_2 that have already been added to δ, since they can be used to process the new fake rule. This is done by putting those transitions back in *trans* again so that they can be processed later at lines 9 or 16. Notice that their flags are set to true, thus preventing them to be processed with push rules again (by the loop guard in line 11).

At line 16 we know that (q, γ, Q') is a transition of $\mathcal{WA}_{pre^*}$ and $\langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q, \gamma \rangle\} \cup R$ is an alternating rule. Therefore, we add the fake rule $\langle p_1, \gamma_1 \rangle \hookrightarrow R$, and update its $\mathcal{F}$-value to include the weight of the transition (q, γ, Q') and the set Q'. If the set R contains more than one element, then similar processes can take place, resulting in more fake rules with less elements in the right-hand sets. Line 9 handles the case when the fake rule is non-alternating.

Lemma 3.1 *Algorithm 3.3 terminates.*

Proof: Since Q and Γ are finite sets, δ is a finite set. Also, the procedure `update` cannot modify weights of any transitions infinitely many times, because the semiring is bounded. Hence, the block at lines 8–18 can only be executed finitely many times. Δ' is finite, since Δ is finite and only finitely many rules are added to Δ'. The flags of the transitions added at line 15 are always true, preventing the loop at line 11 to be entered more than once with a transition having the same weight, and therefore the loop can be entered finitely many times. As a result, all loops after line 8 terminate, and only finite number of elements can be added to *trans*. Once weights of transitions in δ can no longer grow, *trans* can no longer grow and will be empty eventually. This causes the algorithm to terminate. $\square$

Lemma 3.2 *Upon termination of Algorithm 3.3, δ and l are equal to the set of transitions and the weight function of $\mathcal{WA}_{pre^*}$, respectively.*

Proof: Let δ_{pre^*} be the set of transitions and l_{pre^*} be the weight function of $\mathcal{WA}_{pre^*}$. We divide the proof into two parts:

"$\subseteq$" We show that throughout the algorithm $\delta \subseteq \delta_{pre^*}$ and for all $t \in \delta$, $l(t) \sqsubseteq l_{pre^*}(t)$ hold. The saturation procedure defines δ_{pre^*} to contain δ_0 and satisfy the saturation rule. Transitions in δ_0 are copied to δ at line 1, and their weights are copied to l at line 2. Since δ contains only elements that are derived from some elements in *trans*, we inspect the lines that change *trans*, and show that all additions to *trans* satisfy the saturation rule:

- At line 1, *trans* is initialized to δ_0.
- Line 5 models the saturation rule in the case of pop rules.
- Line 6 models the saturation rule in the case of rules with empty set as the right-hand sides.
- Line 15 always adds transitions that are already in δ to *trans*, and therefore always satisfies the saturation rule.
- Line 10 processes the transition (q, γ, Q') and the rule $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, which was added to Δ' because of either of the following three reasons:
  1. The rule was added to Δ' at line 3, i.e. $r = \langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle \in \Delta$ and $\mathcal{F}(r) = (f(r), 1', \emptyset)$, and therefore updating the transition (p, γ, Q') with weight $(f(r) \otimes (l(p, \gamma, Q') \odot 1'))$ satisfies the saturation rule.

2. The rule was added to Δ' at line 12, which implies that r' was $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, and the loop at line 11 was entered with some p'',γ' such that $r = \langle p_1, \gamma_1\rangle \hookrightarrow \langle p'', \gamma'\gamma\rangle \in \Delta$ and $(p'', \gamma', q) \in \mathit{trans}$. As a result, $(f(r) \otimes l(p'', \gamma', q), 1', \emptyset)$ was added to $\mathcal{F}(r')$ at line 13. Then, when r' is considered at line 9 because of the transition (q, γ, Q'), we update transition $(p_1, \gamma_1, Q' \cup \emptyset)$ with weight $f(r) \otimes l(p'', \gamma', q) \otimes (l(q, \gamma, Q') \odot 1')$ according to the saturation rule.
3. The rule was added to Δ' at line 17, which implies that the loop at line 16 was entered with some q'_1, γ'_1, Q'_1 such that $r = \langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q, \gamma\rangle\} \cup \{\langle q'_1, \gamma'_1\rangle\} \in \Delta'$ (i.e., $R = \{\langle q'_1, \gamma'_1\rangle\}$) and the transition $(q'_1, \gamma'_1, Q'_1) \in \mathit{trans}$. This information was saved by adding $(a, l(q'_1, \gamma'_1, Q'_1) \odot b, Q'_1 \cup Q'')$ to $\mathcal{F}(\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle)$ for each $(a, b, Q'') \in \mathcal{F}(r)$ at line 18. Now, we consider again where the rule r was added to Δ'. There are three cases:
   (a) If r was added to Δ' at line 3, i.e. $r \in \Delta$, then $\mathcal{F}(r) = \{(f(r), 1', \emptyset)\}$. Updating the transition $(p_1, \gamma_1, Q' \cup Q'_1 \cup \emptyset)$ with weight $f(r) \otimes (l(q, \gamma, Q') \odot l(q'_1, \gamma'_1, Q'_1) \odot 1')$ therefore conforms to the saturation rule.
   (b) If r was added to Δ' at line 12, then $\gamma = \gamma'_1$, i.e. the rule r was $r = \langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q, \gamma\rangle, \langle q'_1, \gamma\rangle\}$. The loop at line 11 must be entered with some p'', γ' s.t. $r_1 = \langle p_1, \gamma_1\rangle \hookrightarrow \langle p'', \gamma'\gamma\rangle \in \Delta$ and $(p'', \gamma', \{q, q'_1\}) \in \mathit{trans}$. As a result, $(f(r_1) \otimes l(p'', \gamma', \{q, q'_1\}), 1', \emptyset)$ was added to $\mathcal{F}(r)$ at line 13. Again, when considering $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, we update $(p_1, \gamma_1, Q' \cup Q'_1)$ with weight $f(r_1) \otimes l(p'', \gamma', \{q, q'_1\}) \otimes (l(q, \gamma, Q') \odot l(q'_1, \gamma'_1, Q'_1) \odot 1')$ according to the saturation rule.
   (c) If r was added to Δ' at line 17, then there must eventually be $r_1 = \langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q, \gamma\rangle, \langle q'_1, \gamma'_1\rangle, \ldots, \langle q'_n, \gamma'_n\rangle\} \in \Delta'$ for some $n \geq 2$ such that this rule was added at either at line 3 or 12, and $(q'_i, \gamma'_i, Q'_i) \in \mathit{trans}$ for all $i \in [n]$. We can perform the analysis similar to (3a) and (3b), and conclude that the update of the transition $(p_1, \gamma_1, Q' \cup \bigcup_{i=1}^n Q'_i)$ with weight $f(r_1) \otimes (l(q, \gamma, Q') \odot \bigodot_{i=1}^n l(q'_i, \gamma'_i, Q'_i))$ conforms to the saturation rule.

"$\supseteq$" We show that upon termination $\delta \supseteq \delta_{\mathit{pre}^*}$ and for all $t \in \delta$, $l(t) \sqsupseteq$

$l_{pre^*}(t)$ hold. Equivalently, we prove that by the time the algorithm terminates, all possible saturation rules have been applied. Two cases are considered:

1. Assume $r = \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$ and there was $p' \xrightarrow{w} Q$ in δ.
   - If $w = \varepsilon$, then $Q = \{p'\}$. The transition (p, γ, p') has been added with weight $f(r)$ in line 5.
   - If $w = \gamma_1$ and $t = (p', \gamma_1, Q) \in \delta$, then (p, γ, Q) has been added with weight $f(r) \otimes l(t)$ in line 10
   - If $w = \gamma_1\gamma_2$ and δ contained transitions $t' = (p', \gamma_1, Q')$ and $t''_j = (q'_j, \gamma_2, Q'_j)$ for every $q'_j \in Q'$, then after t' was examined Δ' contained the rule $r' = \langle p, \gamma \rangle \hookrightarrow \{\langle q'_j, \gamma_2 \rangle \mid q'_j \in Q'\}$ (line 12), and its $\mathcal{F}$-value included $(f(r) \otimes l(t'), 1', \emptyset)$ (line 13). Notice that all t''_j that had been examined before t' were put back to *trans* (line 15), and therefore were inspected again after r' was constructed. At this point, there were two possible cases when a transition t''_j was examined, depending on the number of elements in the right-hand set of the rule.
     (a) If there were more than one element in the set, line 17 added a rule without $\langle q''_j, \gamma_2 \rangle$ in the right-hand side set to Δ'. The weight of t''_j and the set Q'_j were kept in the $\mathcal{F}$-value of the new rule (line 18). This step is repeated for each different t''_j until there is one element in the right-hand-side set.
     (b) If there was one element in the set, i.e. we had $\langle p, \gamma \rangle \hookrightarrow \langle q''_j, \gamma_2 \rangle$ at line 9, then $(p, \gamma, \bigcup_j Q'_j)$ has been updated with $f(r) \otimes l(t') \otimes \bigodot_j l(t''_j)$ at line 10.
2. Assume $r = \langle p, \gamma \rangle \hookrightarrow \{\langle q_1, \gamma_1 \rangle, \ldots, \langle q_n, \gamma_n \rangle\} \in \Delta$ for some $n \geq 2$ and δ contained $t_i = (q_i, \gamma_i, Q_i)$ for all $i \in [n]$. When a transition t_i was examined, Δ' contained the rule identical to r, but without $\langle q_i, \gamma_i \rangle$ in the right-hand set (line 17). The weight of t_i and the set Q_i were saved in the $\mathcal{F}$-value of the new rule (line 18). This step was repeated for other transitions, and when the last transition was examined the transition $(p, \gamma, \bigcup_{i=1}^{n} Q_i)$ has been updated with weight $f(r) \otimes \bigodot_{i=1}^{n} l(t_i)$ at line 10.

□

Algorithm 3.3 may require exponential time, since the number of transitions of $\mathcal{WA}_{pre^*}$ can be exponential in the number of states. However, a closer look at the complexity reveals that the algorithm is exponential only in a proper subset of states, which can be small depending on the instance.

We conduct a careful analysis in terms of certain parameters of the input, which are listed below:

- Δ_a denotes the set of alternating rules, and a denotes the maximum number of elements in their right-hand sets. Δ_0 denotes the set of pop rules *and* rules where the right-hand sets are the empty set. Δ_1 and Δ_2 denote the set of normal and push rules, respectively.
- The set of *pop control locations*, denoted by P_ε, is the set of control locations $p' \in P$ such that Δ_0 contains some rules $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$.
- We define Q_{ni} as the set of non-initial states, i.e. $Q_{ni} = Q \setminus P$.

The following lemma first focuses on the case in which weights are ignored, so that each transition can be added to δ only once. Alternatively, one can think of the semiring $(\{0,1\}, \vee, \wedge, \wedge, 0, 1, 1)$, where 0 and 1 denote false and true, respectively, and the disjunction ($\vee$) and conjunction ($\wedge$) have their usual meanings. We set $f(r) = 1$ for all $r \in \Delta$ and $l_0(t) = 1$ for all $t \in \delta_0$. The lemma can be easily generalized by using the boundedness property of the semiring.

Lemma 3.3 *When weights are ignored, Algorithm 3.3 takes* $O(|\delta_0| + |\Delta_0| + |\Delta_1| 2^n + (|\Delta_2| n + |\Delta_a| 2^a) 4^n)$ *time, where* $n = |P_\varepsilon| + |Q_{ni}|$.

Proof: Let ν be the smallest set of states such that for every transition (q, γ, Q') added to *trans* at any point during the algorithm, Q' only contains states of ν. Because of lines 1 and 5, we have $P_\varepsilon \cup Q_{ni} \subseteq \nu$. However, all other lines that add transitions to *trans* (namely, 10 and 15) do not add more elements to ν, since for every transition (q, γ, Q') they add, either the transition was added to *trans* before, or Q' must be a union of Q'' for some (q', γ', Q'') that were in *trans*. So, $\nu = P_\varepsilon \cup Q_{ni}$.

Because of the definition of ν, the number of sets $Q' \subseteq Q$ for which, after termination of the algorithm, δ contains a transition of the form (q, γ, Q') is $2^{|\nu|} = 2^n$.

We now consider the number of times the statements inside the main loop are executed. Line 12 is executed once for each combination of $\langle p_1, \gamma_1 \rangle \hookrightarrow$

$\langle q, \gamma\gamma_2 \rangle \in \Delta_2$ and (q, γ, Q'), i.e. $O(|\Delta_2|2^n)$ times. The alternating rules inside Δ' only come from Δ_a, line 17, and line 12, and hence there are $O(|\Delta_a|2^a + |\Delta_2|2^n)$ of them. Line 17 is executed once for each combination of alternating rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q, \gamma \rangle\} \cup R \in \Delta'$ and (q, γ, Q'). Therefore, line 17 is executed $O(|\Delta_a|2^{(a+n)} + |\Delta_2|4^n)$ times. Moreover, since Q' and γ_2 are fixed, line 15 is executed $O(|\Delta_2|4^n)$ times.

Line 10 is executed once for each combination of non-alternating rule in Δ' and (q, γ, Q'). Since the size of Δ' of this form is $O(|\Delta_1| + |\Delta_2|n + |\Delta_a|a)$ and $\mathcal{F}(r)$ has at most 2^n elements for each rule $r \notin \Delta_1$, line 10 is executed $O(|\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|a)4^n)$ times.

Initially, the loops at lines 1, 5, and 6 are executed $|\delta_0| + |\Delta_0|$ times. Therefore, the time complexity can be concluded from the number of times the statements in the algorithm are executed: $O(|\delta_0|+|\Delta_0|+|\Delta_1|2^n+(|\Delta_2|n+|\Delta_a|2^a)4^n)$. □

In typical applications, we usually start with a small automaton, i.e. δ_0 and Q_{ni} will be small. In that case n will be dominated by $|P_\varepsilon|$. In this case, the complexity can be simplified to $O(|\Delta_0| + |\Delta_1|2^{|P_\varepsilon|} + (|\Delta_2||P_\varepsilon| + |\Delta_a|2^a)4^{|P_\varepsilon|})$.

Theorem 3.2 *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted alternating pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S}$ is a bounded idempotent semiring having c as the maximal length of ascending chains. Let $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l)$ be a weighted alternating $\mathcal{WP}$-automaton, where $\mathcal{A} = (Q, \Gamma, \delta, P, F)$. There exists a weighted $\mathcal{WP}$-automaton $\mathcal{WA}_{pre^*}$ such that $L(\mathcal{WA}_{pre^*}) = pre^*(L(\mathcal{WA}))$ and for all $c \in L(\mathcal{WA}_{pre^*})$, $F_{\mathcal{WA}_{pre^*}}(c) = \bigoplus_{C' \subseteq L(\mathcal{WA})} T(c, C') \otimes \bigodot_{c' \in C'} F_{\mathcal{WA}}(c')$. Moreover, $\mathcal{WA}_{pre^*}$ can be constructed in $O(c(|\delta_0| + |\Delta_0| + |\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|2^a)4^n))$ time, where $n = |P_\varepsilon| + |Q_{ni}|$.*

Given an alternating pushdown system $\mathcal{P}$, a configuration c of $\mathcal{P}$, and a set of configurations C, the *backward reachability* problem for $\mathcal{P}$, c, and C is to check whether $c \in pre^*(C)$. By theorem 3.2, the problem is in EXPTIME. The following theorem shows a corresponding lower bound. It is a rather straightforward modification of a theorem of [12].

Theorem 3.3 *The backward reachability problem for alternating pushdown systems is* EXPTIME-*complete, even if C is a singleton.*

Proof: We use a reduction from the acceptance problem for alternating Turing machines.

More specifically, let $\mathcal{M} = (Q, \Sigma, \delta, q_0)$ be an alternating Turing machine, where the control states Q are partitioned into *existential*, *universal*, *accepting*, and *rejecting* states, and where $\delta\colon Q \times \Sigma \to 2^{Q\times\Sigma\times\{L,N,R\}}$ is the transition function.

Let us assume that, when started on the input w, $\mathcal{M}$ uses at most $p(|w|)$ space on the tape, where p is some polynomial independent of w. Thus, a configuration of $\mathcal{M}$ can be represented by a word from $\Sigma^* Q\Sigma\Sigma^*$ of length $p(|w|)$. In a configuration uqv, u are the tape contents to the left of the head, q is the current state, and v are the tape contents under and to the right of the head, including blanks for cells that have not yet been visited. The initial configuration for input w is $q_0 w$ (padded with blanks if needed).

The *computation* of $\mathcal{M}$ on an input w is a tree whose nodes are the tape configurations of $\mathcal{M}$, rooted at the initial configuration and where the children of each configuration are its successor configurations (w.r.t. δ). If α is a configuration of $\mathcal{M}$, we denote by T_α the subtree rooted at α. A subtree T_α, where q is the control state of α, is called *accepting* if either

- q is accepting; or
- q is existential and there is a successor β of α such that T_β is accepting; or
- q is universal and for any successor β of α, T_β is accepting.

The problem to check whether the computation of $\mathcal{M}$ on w is accepting is known to be EXPTIME-complete [12].

Given $\mathcal{M}$ and w, we now construct an alternating pushdown system $\mathcal{P}$ and configurations c, c' such that $c \in \mathit{pre}^*_{\mathcal{P}}(\{c'\})$ if and only if the computation of $\mathcal{M}$ on w is accepting. This proves that the backward reachability problem is EXPTIME-hard, and together with Theorem 3.2, EXPTIME-complete.

The stack alphabet of $\mathcal{P}$ is $Q \cup \Sigma \cup \{\#, L, N, R\}$. A run of $\mathcal{P}$ works in two phases. In the first phase, $\mathcal{P}$ begins at a configuration $\langle \mathit{Start}, \# \rangle$, then nondeterministically pushes a word $\#c_0\#t_1\#c_1\#t_2\#c_2\#t_3\cdots$ onto the stack, where (i) c_0 is the initial configuration of $\mathcal{M}$ on w, (ii) $c_1, c_2, \ldots$ are arbitrary configurations of $\mathcal{M}$, and (iii) t_i, for all $i \geq 1$, represents a transition of $\mathcal{M}$ from c_{i-1} to c_i. A transition t_i is constructed as follows: After pushing c_{i-1}, $\mathcal{P}$ first chooses a pair $(q, a) \in Q \times \Sigma$ and writes qa to the stack. Then, if q is accepting, $\mathcal{P}$ goes to a control state *Test* and enters the second phase (see below). If q is existential, $\mathcal{P}$ nondeterministically chooses a triple

(q', a', m) from $\delta(q, a)$, pushes $q'a'm$ to the stack and continues with c_i. If q is universal, and $|\delta(q, a)| = n$, then $\mathcal{P}$ executes an alternating rule with branching degree n, where each branch writes a distinct element of $\delta(q, a)$ to the stack, and then continues with c_i.

Notice that in the first phase, there is no guarantee that c_{i-1}, t_i, and c_i are correctly related to each other. Consider a run of $\mathcal{P}$ where each branch has entered the control state *Test*, and assume (for a moment) that the choices of subsequent configurations and transitions along each branch correctly represent steps in $\mathcal{M}$. Since the branching behavior of the run corresponds to the branching behavior of the transitions chosen along each branch, and each branch has entered an accepting state, such a run is possible if and only if the computation of $\mathcal{M}$ on w is accepting. All that remains is to check (on each branch) whether the configurations and transitions are correctly related to each other. This is done in the second phase.

In the second phase, $\mathcal{P}$ checks (for each branch) whether the following holds:

(i) for each pair c_{i-1}, t_i, where $i \geq 1$ and $t_i = qa\ldots$, c_{i-1} has the form $uqav$ for some $u, v \in \Sigma^*$;

(ii) for each triple c_{i-1}, $t_i = qaq'a'm$, c_i, where $i \geq 1$, the control state in c_i is q', and its position is correct w.r.t. the position of the control state in c_{i-1} and m;

(iii) for each triple c_{i-1}, t_i, c_i, where $i \geq 1$, and each $j \in \{1, \ldots, p(|w|)\}$, the symbol on the j-th tape cell in c_i is correct w.r.t. the j-th tape symbol in c_{i-1} and t_i.

To perform these checks, the second phase can be seen to consist of a ‘popping thread’ that pops the stack contents and forks off a ‘checking thread’ at each position where a check is required. The popping thread can be implemented by an alternating rule of the kind $\langle \mathit{Test}, \gamma \rangle \hookrightarrow \{\langle \mathit{Test}, \varepsilon \rangle, \langle \mathit{Check}, \gamma \rangle\}$, where $\langle \mathit{Test}, \varepsilon \rangle$ is the continuation of the ‘popping thread’, and $\langle \mathit{Check}, \gamma \rangle$ is the beginning of the ‘checking thread’.

The checking thread for condition (i) is simple to implement: the thread pops (q, a) from the stack, enters a control state $\mathit{Check}_1(q, a)$, then removes the configuration c_{i-1} from the stack, checking whether condition (i) is met.

The checking thread for condition (ii) is similar, except for the fact that the thread needs a counter up to $p(|w|)$ to check that the position is correct.

The checking thread for condition (iii) remembers the symbol of c_i at position j and whether the head is at position $j-1$, j, $j+1$, or somewhere else, then removes the rest of c_i and reads t_i. From this information it can conclude which symbol position j in c_{i-1} should have had. It then removes part of c_{i-1} up to position j and checks whether it contains the correct symbol. Again, a counter up to $p(|w|)$ is needed.

Assume that all successful checking threads continue to remove the stack contents and then enter a control state *End*, and that the same holds for the popping thread. Then, all threads become $\langle \mathit{End}, \varepsilon\rangle$ if and only if all branches of $\mathcal{P}$ in the first phase represented correct computations of $\mathcal{M}$. Putting it differently, $\langle \mathit{Start}, \#\rangle \in \mathit{pre}^*(\{\mathit{End}, \varepsilon\})$ if and only if the computation of $\mathcal{M}$ on w is accepting. The number of control states in $\mathcal{P}$ is $O(p(|w|) \cdot |Q| \cdot |\Sigma|)$. $\square$

A special case

Recall from the saturation procedure that the exponential complexity of the algorithm is due to the fact that the target of the new transition can be an arbitrary set of states, and so we may have to add an exponential number of new states in the worst case. We now consider a special class of instances in which a new transition (p, γ, Q) need to be added only if Q is a singleton. We show that a suitable modification of Algorithm 3.3 has polynomial running time.

Given a set of configurations C, we say that $(\mathcal{WP}, C)$ is a *good instance* if for every $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\} \in \Delta$ with $n \geq 2$ and for every $i \in [n]$, $\langle p_i, w_i w\rangle \in \mathit{pre}^*(C)$ implies $w = \varepsilon$.

Intuitively, if the set C can be reached from $\langle p_i, w_i\rangle$, then it cannot be reached from any $\langle p_i, w_i w\rangle$, where w is a nonempty word. Besides, we define a class of alternating pushdown systems which always induces good instances. A *simple* alternating pushdown system is a tuple (P, Γ, Ξ, Δ), where (P, Γ, Δ) is an alternating pushdown system and $\Xi \subseteq \Gamma$ is a set of bottom stack symbols. Moreover, all transition rules in Δ are of the following forms:

- $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$, where $p, p' \in P$, $\gamma \in \Gamma \setminus \Xi$, and $w \in (\Gamma \setminus \Xi)^*$,
- $\langle p, \bot\rangle \hookrightarrow \{\langle p_1, w_1 \bot_1\rangle, \ldots, \langle p_n, w_n \bot_n\rangle\}$, where $\bot, \bot_i \in \Xi$ and $w_i \in (\Gamma \setminus \Xi)^*$ for all $i \in [n]$.

Notice that the applications in Chapter 6 are based on this particular class.

As mentioned above, we introduce the following modification to the saturation rule: a new transition (p, γ, Q) is added only if Q is a singleton.

Theorem 3.4 *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, and C be a good instance, and let $\mathcal{WA}$ be a non-alternating automaton recognizing C. Assume without loss of generality that $\mathcal{WA}$ has one single final state. Then, by applying the following modified saturation procedure:*

$$\begin{array}{l} \textit{if } \langle p, \gamma\rangle \overset{a}{\hookrightarrow} \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\} \textit{ and } p_i \xrightarrow{w_i(b_i)} q \textit{ for all } i \in [n], \\ \textit{add } t = (p, \gamma, q) \textit{ to } \delta \textit{ and update } l(t) = l(t) \oplus (a \otimes \bigodot_{i=1}^{n} b_i), \end{array}$$

the resulting automaton $\mathcal{WA}_{pre^*}$ is non-alternating and recognizes language $pre^*(C)$ such that for each $c \in pre^*(C)$,

$$F_{\mathcal{WA}_{pre^*}}(c) = \bigoplus_{c' \in C} T(c, c') \otimes F_{\mathcal{WA}}(c') \ .$$

Proof: Because of the modification in the rule, the modified saturation procedure never adds an alternating rule, and so it yields a nondeterministic automaton.

We claim that if $\mathcal{WA}_{pre^*}$ contained a transition $(p, \gamma, \bigcup_{i=1}^{n} Q_i)$ such that $\bigcup_{i=1}^{n} Q_i$ is not a singleton, then at least one Q_i, where $i \in [n]$, would contain a redundant state, i.e. a state from which no word can be accepted. It follows that the transition need not be added.

To prove the claim, observe that $(p, \gamma, \bigcup_{i=1}^{n} Q_i)$ is obtained from some rule $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\} \in \Delta$ and a set of paths $p_1 \xrightarrow{w_1} Q_1, \ldots, p_n \xrightarrow{w_n} Q_n$. Let q_f be the final state of $\mathcal{WA}$. If $Q_i \neq \{q_f\}$ for some $i \in [n]$, then Q_i contains a non-final state q of $\mathcal{WA}$. If q were non-redundant in $\mathcal{WA}_{pre^*}$, then $\mathcal{WA}_{pre^*}$ would recognize a word $p_i w_i w$ where $w \neq \varepsilon$. But then $p_i w_i w \in pre^*(C)$, contradicting the assumption that $(\mathcal{WP}, C)$ is a good instance. $\square$

To construct $\mathcal{WA}_{pre^*}$ we again impose without loss of generality restrictions (R1) and (R2), and for every rule $\langle p, \gamma\rangle \hookrightarrow R \in \Delta : |R| \leq 2$. Algorithm 3.3 implements the modified saturation procedure after the following change to line 9:

forall $r = \langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle \in \Delta'$ **and** $(a, b, Q'') \in \mathcal{F}(r)$ s.t. $Q'' \cup Q' = Q'$

Lemma 3.4 *When weights are ignored, the modified Algorithm 3.3 takes $O(|\delta_0| + |\Delta_0| + (|\Delta_1| + |\Delta_a|)n + |\Delta_2|n^2)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$, when applied to a good instance.*

Proof: The proof is similar to Lemma 3.3. Let ν be the set of states such that for every transition $(q, \gamma, q') \in \mathit{trans}$ at any point of the algorithm, $q' \in \nu$. Lines 1–6 add $|P_\varepsilon| + |Q_{ni}|$ elements to ν. Since no other lines add more elements to ν, the size of ν is $|P_\varepsilon| + |Q_{ni}| = n$ at the end of the algorithm. Therefore, the resulting δ contains at most n possible states in its right-hand side.

Line 12 is executed $O(|\Delta_2|n)$ times, therefore adding $O(|\Delta_2|n)$ rules to Δ'. Line 17 considers alternating rules, which can only come from lines 3, so they contribute $O(|\Delta_a|)$ elements to Δ'. Line 15 is executed $O(|\Delta_2|n^2)$ times, since q', γ_2 are fixed. Also, the modification of line 9 causes line 10 to be executed $O(|\Delta_1|n + |\Delta_2|n^2 + |\Delta_a|n)$ times. Altogether, the statements in the algorithm cannot be executed more than $O(|\delta_0| + |\Delta_0| + (|\Delta_1| + |\Delta_a|)n + |\Delta_2|n^2)$ times, and this leads to the time complexity. $\square$

Note that Algorithm 3.3, when applied to a non-alternating pushdown system (i.e. one with $\Delta_a = \emptyset$), has the same complexity as the algorithm from [21] that was specially designed for non-alternating pushdown systems.

3.1.3 Pushdown networks

Throughout this section, let $\mathcal{WN} = (\mathcal{N}, \mathcal{S}, (f_i)_{i \in [n]})$ denote a fixed weighted pushdown network, where $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i \in [n]})$. Given a global configuration C, the *global reachability problem* is to compute the set

$$\mathit{post}^*(C) = \{C' \mid T(C, C') \neq 0\} .$$

However, the global reachability problem is undecidable; more precisely, it is in general undecidable whether $C' \in \mathit{post}^*(C)$, for a given pair C, C' [50]. For this reason, one tries to approximate $\mathit{post}^*(C)$. One such approximation, introduced in [47], uses the notion of *context-bounded* computations.

A *context* of $\mathcal{WN}$ is a sequence of transitions where all moves are made by a single process. In other words, let us define a global reachability relation of the i-th process by overloading the symbol $\Rightarrow_i$ as follows

$$\langle g, w_1, \ldots, w_i, \ldots, w_n \rangle \overset{a}{\Rightarrow}_i \langle g', w_1, \ldots, w_i', \ldots, w_n \rangle \text{ iff } \langle g, w_i \rangle \overset{a}{\Rightarrow}_i \langle g', w_i' \rangle .$$

Then $\Rightarrow_i$ is a relation between global configurations reachable from each other in a single context. Given two configurations C and C', we define

$$T_i(C, C') = \bigoplus \{a \in D \mid C \overset{a}{\Rightarrow}_i C'\} .$$

to be the traces of all runs of the i-th process starting from C and reaching C'. Correspondingly, we define the *local reachability problem* is to compute $\overline{post}_i^*(C) = \{C' \mid T_i(C, C') \neq 0\}$, i.e. $\overline{post}_i^*(C)$ is the set of global configurations reachable from C by moves of the i-th process. Moreover, we define $\twoheadrightarrow_j$, where $j \geq 0$, the global reachability relation within j contexts as follows:

- $C \xrightarrow{1}\!\!\!\!\twoheadrightarrow_0 C$, for all global configurations C, and
- if $C \xrightarrow{a}\!\!\!\!\twoheadrightarrow_j C'$ and $C' \overset{b}{\Rightarrow}_i C''$ for some $i \in [n]$, then $C \xrightarrow{a \otimes b}\!\!\!\!\twoheadrightarrow_{j+1} C''$.

The traces of all runs within j contexts are defined as:

$$T_{\leq j}(C, C') = \bigoplus \{a \in D \mid C \xrightarrow{a}\!\!\!\!\twoheadrightarrow_j C'\}$$

We can now define the central problem of this section: Given $k \geq 1$ and an initial configuration C, the *context-bounded reachability problem* is to compute the set of configurations reachable in at most k contexts, i.e. the set

$$post^*{}_{\leq k}(C) = \{C' \mid T_{\leq k}(C, C') \neq 0\} \ .$$

View tuples

In addition to $\mathcal{WN}$, let us fix a global configuration C and a context bound $k \geq 1$ for the rest of the section.

The principal problem that one faces when solving the context-bounded reachability problem is to find a data structure for representing the set $post^*{}_{\leq k}(C)$. Note that while the global storage can assume only finitely many values, the number of possible stack contents is infinite, thus finding a suitable data structure for representing sets of global configurations is not straightforward. Here, we define a data structure that will be helpful to discuss the algorithm in this section. The main idea is to represent $post^*{}_{\leq k}(C)$ by so-called *view tuples*.

Let $C' = \langle g, w_1, \ldots, w_n \rangle$ be a global configuration. For $i \in [n]$, we call the local configuration $\langle g, w_i \rangle$ the *i-th view* of C'. A *view tuple* $T = (V_1, \ldots, V_n)$ is a collection where V_i is a regular set of configurations, i.e. a set of i-th views for each $i \in [n]$, represented by a weighted automaton (see Section 3.1.1). T is associated with the following set of configurations:

$$[\![T]\!] = \{\langle g, w_1, \ldots, w_n \rangle \mid \langle g, w_i \rangle \in V_i \text{ for all } i \in [n]\} \ .$$

Not every set of global configurations can be represented as a view tuple. As a running example, let us consider a system with two processes, globals g, g', g'' and stack alphabet a, b. Consider the set of global configurations $\mathcal{C} = \{\langle g, a, a\rangle, \langle g', b, a\rangle, \langle g'', a, a\rangle, \langle g'', b, b\rangle\}$. Suppose that there is a view tuple $T = (V_1, V_2)$ such that $[\![T]\!] = \mathcal{C}$. Then V_1 necessarily contains the pair $\langle g'', a\rangle$ and V_2 the pair $\langle g'', b\rangle$. But then, $[\![T]\!]$ also contains $\langle g'', a, b\rangle$, which is not in $\mathcal{C}$.

More importantly, the sets arising in the context-bounded reachability problem are not representable as view tuples. Ignoring weights, suppose from the example above that $\Delta_1 = \{\langle g, a\rangle \hookrightarrow \langle g', b\rangle\}$ and $\Delta_2 = \{\langle g, a\rangle \hookrightarrow \langle g'', a\rangle, \langle g', a\rangle \hookrightarrow \langle g'', b\rangle\}$. Then, $\mathit{post}^*_{\leq 2}(\langle g, a, a\rangle)$ is exactly the set $\mathcal{C}$ above.

In general, the result of a context-bounded reachability query is only representable as a *union* of view tuples. For instance $\mathcal{C}$ can be partitioned into the sets $\mathcal{C}_1 := \{\langle g, a, a\rangle, \langle g'', a, a\rangle\}$ and $\mathcal{C}_2 := \{\langle g', b, a\rangle, \langle g'', b, b\rangle\}$, which are both representable as view tuples. As we shall see, there are different ways to choose the view tuples contained in this union, and we aim to find one which requires few tuples.

A meta-algorithm for context-bounded reachability

In this section we discuss a meta-algorithm to solve the context-bounded reachability problem. Two similar algorithms were proposed in [47] and [7]. While they differ in some details, the idea can be summarized by Algorithm 3.4. It can be characterized as a worklist algorithm that computes the effect of one context at a time.

The entries of the worklist are triples (j, i, T), where T is a view tuple reachable within j contexts such that i was the process that made the last move ($i = 0$ iff $j = 0$). Initially, the worklist contains just one view tuple representing the initial configuration C. In each iteration, the algorithm picks a view tuple from the worklist and computes the configurations that can be reached through a single additional context. Notice that since we are dealing with regular sets of configurations, this can be done by solving the local reachability problem, i.e. the reachability problem for pushdown systems, see Section 3.1.1. The previously active process, i, is excluded from consideration because it would not add any new information.

The result of the local reachability algorithm is denoted by P, and the principal problem is that P may no longer be representable as a single view tuple. The task of the split function in line 9 is to generate a set of view

Input: Weighted pushdown network $\mathcal{WN} = (\mathcal{N}, \mathcal{S}, (f_i)_{i\in[n]})$, where $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i\in[n]})$, initial configuration $(g, w_1, \ldots, w_n)$, context bound k.

Output: The set of reachable global configurations.

```
1  $result := \emptyset$;
2  $worklist := \{ (0, 0, (\{(g, w_1)\}, \ldots, \{(g, w_n)\})) \}$;
3  while $worklist \neq \emptyset$ do
4      remove $(j, i, T)$ from $worklist$;
5      add $[\![T]\!]$ to $result$;
6      if $j < k$ then
7          forall $i' \in [n] \setminus \{i\}$ do
8              $P := \overline{post}^*_{i'}([\![T]\!])$;
9              forall $T' \in split(P)$ do
10                 add $(j + 1, i', T')$ to $worklist$;
11 return $result$;
```

Algorithm 3.4: A context-bounded reachability algorithm

tuples such that $\bigcup_{T'\in split(P)} [\![T']\!] = P$. In [47, 7], split works as follows:

$$\begin{aligned} split(P) &= \{T_g \mid g \in G\}, \text{ where} \\ T_g &= P \cap \{(g, w_1, \ldots, w_n) \mid w_i \in \Gamma^*, i \in [n]\} . \end{aligned}$$

It can be shown that the resulting sets are always view tuples. However, after each context, every worklist entry is split $|G|$ different ways. We call this approach *eager splitting*. Loosely speaking, eager splitting processes $n^k|G|^k$ worklist entries. Moreover, after each split the algorithm will consider every element of G individually, which does not lend itself to a meaningful symbolic implementation (e.g., using efficient set representations such as BDDs).

The rest of this section identifies a coarser partition of P that leads to fewer splits, in the hope of making the algorithm faster in practice. We call this approach *lazy splitting*.

Lazy splitting

To simplify the presentation we consider the case of two processes and assume without loss of generality that the second process is active, i.e. given a view tuple $T = (V_1, V_2)$, the task is to (i) compute the set $\overline{post}_2^*([\![T]\!])$ and (ii) split this set into new view tuples. Recall that a global configuration of pushdown network with two processes is a tuple (g, w_1, w_2), where g is a global and w_i is a local configuration of the i-th process.

Throughout this section we identify a set $X \subseteq X_1 \times \ldots \times X_n$ and the predicate $X(x_1, \ldots, x_n)$ such that $X(a_1, \ldots, a_n)$ holds iff $(a_1, \ldots, a_n) \in X$. We liberally mix set and logical notations, and write e.g. $A(x) = \exists\, y\colon B(x, y)$ to mean $A = \{\, x \mid \exists y\colon B(x, y)\,\}$. Abusing notation, we shall sometimes denote the set $[\![T]\!]$, where T is a view tuple, simply by T.

We proceed as follows: We first identify a property between globals (called *confluence*) that prevents certain configurations from being included in the same partition. We then show how the confluence relation can be computed symbolically, and finally how partitions can be computed from this relation.

Confluence and safe partitions Let $R_2(g, w, g', w')$ be the reachability predicate for the second thread, i.e., $R_2(g, w, g', w')$ holds iff $\langle g, w\rangle \overset{a}{\Rightarrow}_2 \langle g', w'\rangle$ for some $a \neq 0$. (As usual, we use unprimed variables for the initial configuration and primed ones for the final configuration.) Using standard logical manipulations we obtain

$$\overline{post}_2^*(T)(g', w_1, w_2') = \exists g : \Big(V_1(g, w_1) \wedge \underbrace{\exists w_2 : V_2(g, w_2) \wedge R_2(g, w_2, g', w_2')}_{=:U_2(g, g', w_2')} \Big) \ .$$

Since g is existentially quantified, $\overline{post}_2^*(T)$ is not always a view tuple. We present a generic approach for representing it as a union of view tuples. The approach is parameterized by a partition of G. The partition is, on the other hand, based on the following property of global values.

Two distinct global values $g_a, g_b \in G$ are *confluent* if there exist g', w'_{2a}, w'_{2b} such that $U_2(g_a, g', w'_{2a})$ and $U_2(g_b, g', w'_{2b})$ hold. A partition of G is *safe* if none of its sets contains two confluent values. Intuitively, two values in the same set of a safe partition cannot be transformed by the second thread into the same value. For instance, let us return to the running example. If we choose T such that $[\![T]\!] = \{\langle g, a, a\rangle, \langle g', b, a\rangle\}$, then $\overline{post}_2^*(T) = \mathcal{C}$ because $\langle g, a, a\rangle \Rightarrow_2 \langle g'', a, a\rangle$ and $\langle g', b, a\rangle \Rightarrow_2 \langle g'', b, b\rangle$. In other words we have $U_2 = \{\langle g, g, a\rangle, \langle g', g', a\rangle, \langle g, g'', a\rangle, \langle g', g'', b\rangle\}$. Therefore, g and g' are confluent, and any safe partition must keep these two values apart.

Notice that safe partitions always exist, because the partition that splits G into singletons is always safe. However, finding a coarser safe partition is not necessarily straightforward, since U_2 may contain infinitely many tuples, and we will show how to deal with this problem later. For the time being, it suffices to point out that *any* safe partition can be used to represent $\overline{post}_2^*(T)$ as a union of view tuples. Let $G_1, \ldots, G_m$ be a safe partition of G. We define sets $V'_{11}, \ldots, V'_{1m}$ of 1-views and sets $V'_{21}, \ldots, V'_{2m}$ of 2-views as follows:

$$V'_{1j}(g', w_1) = \exists g : V_1(g, w_1) \wedge G_j(g) \wedge \exists w'_2 : U_2(g, g', w'_2) \quad (3.1)$$
$$V'_{2j}(g', w'_2) = \exists g : U_2(g, g', w'_2) \wedge G_j(g) \quad (3.2)$$

Intuitively, V'_{1j} contains the local configurations of the first thread for which the second thread can reach the local configuration w'_2 while leaving the global variable in state g'. Therefore, if the first thread initially has $\langle g, w_1 \rangle$ as 1-view, it ends with $\langle g', w_1 \rangle$: the local configuration w_1 has not changed, but the value of the global variable has. The intuition behind V'_{2j} is similar.

In the example above, we could choose $G_1 = \{g, g''\}$ and $G_2 = \{g'\}$ as a safe partition. Under this assumption the view tuples (V'_{11}, V'_{21}) and (V'_{12}, V'_{22}) as defined above would represent the previously defined sets $\mathcal{C}_1$ and $\mathcal{C}_2$, whose union is indeed equal to $\mathcal{C}$. The following theorem states that this works for *every* safe partition.

Theorem 3.5 *Let $\{V'_{1j}\}_{j \in [m]}$ and $\{V'_{2j}\}_{j \in [m]}$ be defined as in (3.1) and (3.2). Then*

$$\overline{post}_2^*(T)(g', w_1, w'_2) = \bigvee_{j=1}^{m} \Big(V'_{1j}(g', w_1) \wedge V'_{2j}(g', w'_2) \Big) .$$

Proof:
($\Rightarrow$): $\overline{post}_2^*(T)(g', w_1, w'_2)$

$$\begin{aligned}
&= \exists g : V_1(g, w_1) \wedge U_2(g, g', w'_2) && \text{(def. of } \overline{post}_2^*(T)) \\
&= \exists g : V_1(g, w_1) \wedge U_2(g, g', w'_2) \wedge \exists j \in [m] : G_j(g) \\
&= \exists j \in [m] : V'_{1j}(g', w_1) \wedge V'_{2j}(g', w'_2) && \text{(logic, def. of } V'_{1j}, V'_{2j}) \\
&\Rightarrow \bigvee_{i=1}^{m} \Big(V'_{1j}(g', w_1) \wedge V'_{2j}(g', w'_2) \Big)
\end{aligned}$$

($\Leftarrow$): Let (g', w_1, w'_2) be a triple satisfying $V'_{1j}(g', w_1) \wedge V'_{2j}(g', w'_2)$ for some $j \in [m]$. By the definition of V'_{1j} and V'_{2j} there exist g_a, g_b, and w''_2 such that $V_1(g_a, w_1)$, $G_j(g_a)$, $U_2(g_a, g', w_2'')$, $U_2(g_b, g', w_2')$, and $G_j(g_b)$ hold. So, g_a and g_b belong to the same set of the partition of G, namely G_j. Furthermore, since $U_2(g_a, g', w_2'')$, $U_2(g_b, g', w_2')$, it follows from the definition of safe partition that g_a and g_b are either confluent or equal. Since the partition used to construct $\{V'_{1j}\}_{j\in[m]}$ and $\{V'_{2j}\}_{j\in[m]}$ is safe, we get $g_a = g_b$. So, in particular, $U_2(g_a, g', w_2')$ holds, which together with $V_1(g_a, w_1)$ implies $\overline{post_2^*}(T)(g', w_1, w_2')$. $\square$

Computing the confluence relation In this part, we show how to compute the relation $C(x, y)$ of confluent pairs x, y symbolically. By its definition,

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g', w'_{2a}, w'_{2b} : U_2(g_a, g', w'_{2a}) \wedge U_2(g_b, g', w'_{2b}) \ .$$

Let us recall the definitions of $U_2(g, g', w_2')$ and $post_2^*(V_2)(g', w_2')$:

$$\begin{aligned} U_2(g, g', w_2') &= \exists w_2 : V_2(g, w_2) \wedge R_2(g, w_2, g', w_2') \\ post_2^*(V_2)(g', w_2') &= \exists g, w_2 : V_2(g, w_2) \wedge R_2(g, w_2, g', w_2') \ . \end{aligned}$$

We now reduce the computation of U_2 to a local reachability problem w.r.t. a modified pushdown system $(G \times G, \Gamma, \Delta_2')$. In other words, we change the system by duplicating the globals. Moreover, we have $\langle(\bar{g}, g), \gamma\rangle \hookrightarrow \langle(\bar{g}, g'), w\rangle$ in Δ_2' iff $\langle g, \gamma\rangle \hookrightarrow \langle g', w\rangle$ in Δ_2, i.e. the value of the first copy is never changed by any transition rule. The reachability relation for the second thread of the modified system is given by $\overline{R}_2((\bar{g}, g), w_2, (\bar{g}', g'), w_2') = R_2(g, w_2, g', w_2') \wedge \bar{g} = \bar{g}'$. Define $\overline{V}_2((\bar{g}, g), w_2) = V_2(g, w_2) \wedge \bar{g} = g$. We have:

$$\begin{aligned} U_2(g, g', w_2') &= \exists w_2 : V_2(g, w_2) \wedge R_2(g, w_2, g', w_2') \\ &= \exists w_2 : \overline{V}_2((g, g), w_2) \wedge \overline{R}_2((g, g), w_2, (g, g'), w_2') \\ &= \exists \bar{\bar{g}}, \bar{g}, w_2 : \overline{V}_2((\bar{\bar{g}}, \bar{g}), w_2) \wedge \overline{R}_2((\bar{\bar{g}}, \bar{g}), w_2, (g, g'), w_2') \\ &= post_2^*(\overline{V_2})((g, g'), w_2') \ . \end{aligned}$$

We turn to the question how to compute C from the automaton representing U_2. For this, let us define $U_2'(g, g') := \exists w \colon U_2(g, g', w)$. Therefore,

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g' \colon U_2'(g_a, g') \wedge U_2'(g_b, g') \ .$$

The modified pushdown system defined above has $G \times G$ as its set of globals. Thus, the symbolic automaton for U_2 uses $G \times G$ as initial states, and a configuration $\langle(g, g'), w\rangle$ is accepted if, starting at state (g, g'), the automaton can read the input w and end up in a final state. Thus, $U_2'(g, g')$ holds if some input is accepted from the state (g, g').

Computing a safe partition Given the confluence relation C, our final goal now is to compute a safe partition. Notice that a partition is safe if and only if its sets are cliques of $\neg C$, the complement of C. Since finding a minimal partition into cliques of a given graph is NP-complete, we restrict ourselves to finding a reasonably coarse safe partition in a symbolic manner. The resulting performance of the reachability algorithm is evaluated in Chapter 5.

```
**Input**: Confluence relation $C(x, y)$, total order $L(x, y)$
**Output**: A safe partition $G_1, \ldots G_m$ of $G$
$S(x, y) := \neg C(x, y)$; $j := 0$;
while $S \neq \emptyset$ do
    pick $(x_0, y_0)$ from $S$;
    $F(x) := S(x, y_0)$;
    while true do
        $D(x, y) := L(x, y) \wedge F(x) \wedge F(y) \wedge \neg S(x, y)$;
        exit if $D = \emptyset$;
        $F(x) := F(x) \wedge \neg(\exists y : D(x, y))$
    $j := j + 1$;
    $G_j(x) := F(x)$;
    $S(x, y) := S(x, y) \wedge \neg F(x) \wedge \neg F(y)$;
```

Algorithm 3.5: An algorithm for computing equivalence classes

Algorithm 3.5 shows the computation of the partition. Its inputs are the confluence relation C and an arbitrary total order relation L on globals. The algorithm repeatedly computes sets of the partition. The inner loop makes sure that F is a clique of S when exiting the loop. D contains the confluent pairs (x, y) of $F \times F$ such that x is smaller than y with respect to the order L. If $D = \emptyset$ then F is a clique. Otherwise, for each $(x, y) \in D$ we remove x. The rôle of L is to guarantee that D is antisymmetric, and so that if x and y are confluent we remove exactly one of them from F.

Notice that the algorithm only uses Boolean operations and existential quantification, and can therefore be easily implemented in a BDD library, given BDD representations of L and C. The computation of C was presented in previous section, and a BDD representation for $L \subseteq G \times G$ is trivial to generate, because by assumption the set G is finite, and any total order (e.g. some lexicographical ordering based on the BDD variables) will do.

Finally, equations (3.1) and (3.2) only use G_j, V_1, U_2, which are all representable as BDDs or as symbolic automata, connected by Boolean operations. Thus, the new view tuples can be obtained by standard operations on BDDs and automata.

3.2 General semirings

We consider in this section another approach for analyzing reachability in pushdown models when weights are neither bounded nor idempotent. Later on, we show that the analyses boil down to solving equation systems over (extended) semirings. We first define equation systems over semirings. The definition can be straightforwardly generalized to extended semirings.

Given a semiring $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$, a set of variable V, the set of *terms* over $\mathcal{S}$ and V, denoted by $\mathcal{S}(V)$ is given by:

$$t ::= a \mid v \mid t \oplus t \mid t \otimes t \ ,$$

where $a \in D$ and $v \in V$. Given n variables $\mathbf{v} = (v_1, \ldots, v_n)$, a term $t_i(\mathbf{v})$ ($i \in [n]$) defines a mapping from D^n to D. An *equation system* is given by $\mathbf{v} = \mathbf{t}(\mathbf{v})$ such that $\mathbf{t}(\mathbf{v}) = (t_1(\mathbf{v}), \ldots, t_n(\mathbf{v}))$.

Solving equation systems over semirings is out of scope of this thesis. We only note here that when $\mathcal{S}$ is ω-continuous, it can be proved that terms in $\mathcal{S}(V)$ define monotonic mappings, i.e. for every term $t(\mathbf{v})$ and $\mathbf{a}, \mathbf{b} \in D^n$, if $\mathbf{a} \sqsubseteq \mathbf{b}$, then $t(\mathbf{a}) \sqsubseteq t(\mathbf{b})$, where the relation $\sqsubseteq$ is pointwise extended. By the theorem of Knaster-Tarski, the least solution of $\mathbf{v} = \mathbf{t}(\mathbf{v})$ exists. Moreover, let $\mathbf{0}$ be a vector of zeros of length n, by Kleene's theorem the completeness property implies that this solution is the supremum of the sequence $(t^n(\mathbf{0}))_{n \in \mathbb{N}}$, which is equal to $\bigoplus_{n \in \mathbf{N}} t^n(\mathbf{0})$ by the ω-continuity property.

The application in Section 6.2 only focuses on the extended semiring $([0,1], +, \cdot, \cdot, 0, 1, 1)$, i.e. terms are polynomials over reals between 0 and 1. In this case, solutions can be effectively estimated by using e.g. Newton's method (see [44] for a reference).

3.2.1 Pushdown systems

Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$, denote a fixed weighted pushdown system. Again, we assume without loss of generality that for every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$, we have $|w| \leq 2$.

We show in this section that the set of traces of all runs leading from given configurations to other configurations can be characterized as the least fixpoint of a system of equations. Without loss of generality, we assume that the initial configurations are of the form $\langle p, \gamma \rangle$, and that all target configurations are of the form $\langle q, \varepsilon \rangle$, where $p, q \in P$ and $\gamma \in \Gamma$, i.e. the initial stack contains a single stack symbol and the target stack is empty. For all p, γ, q, we define the variables as triples $[p, \gamma, q]$, and the set of equations as follows:

$$\begin{array}{rcll} [p,\gamma,q] & = & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle q,\varepsilon\rangle} & a \\ & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\rangle} & a \otimes [p',\gamma',q] \\ & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\gamma''\rangle} & a \otimes \Big(\bigoplus_{p''\in P} [p',\gamma',p''] \otimes [p'',\gamma'',q] \Big) \end{array} \tag{3.3}$$

Intuitively, equation (3.3) lists the traces of all runs starting from $\langle p, \gamma \rangle$ and reaching $\langle q, \varepsilon \rangle$. Since the runs always start form the configuration $\langle p, \gamma \rangle$, we only consider the rules with $\langle p, \gamma \rangle$ on their left-hand side. There are three types of runs depending on rules that are first executed:

1. if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle q, \varepsilon \rangle$, then $\langle p, \gamma \rangle \overset{a}{\Rightarrow} \langle q, \varepsilon \rangle$,

2. if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma' \rangle$, then $\langle p, \gamma \rangle \overset{a}{\Rightarrow} \langle p', \gamma' \rangle \xRightarrow{[p',\gamma',q]} \langle q, \varepsilon \rangle$, and

3. if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\gamma'' \rangle$, then $\langle p, \gamma \rangle \overset{a}{\Rightarrow} \langle p', \gamma'\gamma'' \rangle \xRightarrow{[p',\gamma',p'']} \langle p'', \gamma'' \rangle \xRightarrow{[p'',\gamma'',q]} \langle q, \varepsilon \rangle$, for all $p'' \in P$.

We prove in the following theorem that Equation (3.3) combines the traces of all possible runs. Recall from Section 2.2.1 that given two configurations c and c', $T(c, c')$ denotes the trace of all runs starting from c and reaching c'.

Theorem 3.6 *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$. The least fixpoint of the system of equations (3.3) always associates $[p, \gamma, q]$ to $T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$, for all $p, q \in P$ and $\gamma \in \Gamma$.*

Proof: Let μ be the least fixed point. We write $[p, \gamma, q]_\mu$ to denote the component of μ which corresponds to the variable $[p, \gamma, q]$.

It is easy to see that the tuple of all $T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$ forms a solution of the equation system. This is done by partitioning all possible runs into disjoint subsets as discussed above. Thus, $[p,\gamma,q]_\mu \sqsubseteq T(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle)$ for all $p,q \in P$ and $\gamma \in \Gamma$.

To prove that $T(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) \sqsubseteq [p,\gamma,q]_\mu$, we first define the relation $\Rightarrow_{\leq k}$ to be the reachability relation involving only runs of lengths less than or equal to k. Formally, $\Rightarrow_{\leq} \subseteq (P \times \Gamma^*) \times D \times \mathbb{N} \times (P \times \Gamma^*)$ is the smallest relation such that

- $c \overset{1}{\Rightarrow}_{\leq 0} c$, for all $c \in P \times \Gamma^*$,
- if $\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',w\rangle$ and $\langle p',ww'\rangle \overset{b}{\Rightarrow}_{\leq k} c$ for some $w' \in \Gamma^*$, $b \in D$, and $c \in P \times \Gamma^*$, then $\langle p,\gamma w'\rangle \overset{a\otimes b}{\Longrightarrow}_{\leq k+1} c$.

We correspondingly define the traces for each $k \in \mathbb{N}$:

$$T_{\leq k}(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) = \bigoplus\{a \mid \langle p,\gamma\rangle \overset{a}{\Rightarrow}_{\leq k} \langle q,\varepsilon\rangle\} \ .$$

Clearly, $T(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) = \bigcup_{k\geq 0} T_{\leq k}(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle)$. Therefore, showing that $T(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) \sqsubseteq [p,\gamma,q]_\mu$ boils down to proving $T_{\leq k}(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) \sqsubseteq [p,\gamma,q]_\mu$ for each $k \in \mathbb{N}$. We proceed by induction on k.

The base case, where $k = 0$, follows immediately from the definition. From the definitions of $T_{\leq k+1}(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle)$ and $[p,\gamma,q]_\mu$ we have

$$\begin{array}{rcll}
T_{\leq k+1}(\langle p,\gamma\rangle,\langle q,\varepsilon\rangle) & = & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle q,\varepsilon\rangle} & a \\
 & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\rangle} & a \otimes \bigoplus\{b \mid \langle p',\gamma'\rangle \overset{b}{\Rightarrow}_{\leq k} \langle q,\varepsilon\rangle\} \\
 & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\gamma''\rangle} & a \otimes \bigoplus\{b \mid \langle p',\gamma'\gamma''\rangle \overset{b}{\Rightarrow}_{\leq k} \langle q,\varepsilon\rangle\}
\end{array}$$

and

$$\begin{array}{rcll}
[p,\gamma,q]_\mu & = & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle q,\varepsilon\rangle} & a \\
 & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p,\gamma'\rangle} & a \otimes [p',\gamma',q]_\mu \\
 & \oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p,\gamma'\gamma''\rangle} & a \otimes \bigoplus_{p''\in P} [p',\gamma',p'']_\mu \otimes [p'',\gamma'',q]_\mu \ .
\end{array}$$

So, it suffices to show

$$\begin{array}{rcl}
\bigoplus\{b \mid \langle p',\gamma'\rangle \overset{b}{\Rightarrow}_{\leq k} \langle q,\varepsilon\rangle\} & \sqsubseteq & [p',\gamma',q]_\mu \text{ and} \\
\bigoplus\{b \mid \langle p',\gamma'\gamma''\rangle \overset{b}{\Rightarrow}_{\leq k} \langle q,\varepsilon\rangle\} & \sqsubseteq & \displaystyle\bigoplus_{p''\in P} [p',\gamma',p'']_\mu \otimes [p'',\gamma'',q]_\mu \ .
\end{array}$$

Since

$$\bigoplus\{b \mid \langle p', \gamma' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle q, \varepsilon \rangle\} = T_{\leq k}(\langle p', \gamma' \rangle, \langle q, \varepsilon \rangle) \ ,$$

by the induction hypothesis we have

$$\bigoplus\{b \mid \langle p', \gamma' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle q, \varepsilon \rangle\} \sqsubseteq [p', \gamma', q]_\mu \ .$$

Furthermore, from the definition of traces and the induction hypothesis

$$\begin{aligned} &\bigoplus\{b \mid \langle p', \gamma'\gamma'' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle q, \varepsilon \rangle\} \\ &\quad \sqsubseteq \quad \textstyle\bigoplus_{p'' \in P} T_{\leq k}(\langle p', \gamma' \rangle, \langle p'', \varepsilon \rangle) \otimes T_{\leq k}(\langle p'', \gamma'' \rangle, \langle q, \varepsilon \rangle) \\ &\quad \sqsubseteq \quad \textstyle\bigoplus_{p'' \in P} [p', \gamma', p'']_\mu \otimes [p'', \gamma'', q]_\mu \ . \end{aligned}$$

Consequently, we conclude that $T_{\leq k+1}(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle) \sqsubseteq [p, \gamma, q]_\mu$. □

3.2.2 Alternating pushdown systems

Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$ denote a fixed weighted alternating pushdown system. Again, we assume without loss of generality that $\mathcal{WP}$ satisfies the restrictions (R1) and (R2) in Section 3.1.2 and, furthermore, for every rule $\langle p, \gamma \rangle \hookrightarrow R \in \Delta : |R| \leq 2$.

We show that the set of traces of all runs leading from given configurations to *sets* of configurations can be characterized as the least fixpoint of a system of equations. Without loss of generality, we assume that the initial configurations are of the form $\langle p, \gamma \rangle$, and that all configurations in the target sets are of the form $\langle q, \varepsilon \rangle$, where $p, q \in \mathcal{P}$ and $\gamma \in \Gamma$, i.e. the initial stack contains a single stack symbol and the target stacks are empty. For all $p \in P, \gamma \in \Gamma, Q \subseteq P$, we define the variables as triples $[p, \gamma, Q]$, and the set of equations as follows:

$$\begin{aligned} [p, \gamma, Q] \quad = \quad & \\ & \textstyle\bigoplus_{\langle p,\gamma \rangle \overset{a}{\hookrightarrow} \emptyset \ \wedge \ Q = \emptyset} \quad a \\ \oplus \quad & \textstyle\bigoplus_{\langle p,\gamma \rangle \overset{a}{\hookrightarrow} \langle p',\varepsilon \rangle \ \wedge \ Q = \{p'\}} \quad a \\ \oplus \quad & \textstyle\bigoplus_{\langle p,\gamma \rangle \overset{a}{\hookrightarrow} \langle p',\gamma' \rangle} \quad a \otimes [p', \gamma', Q] \\ \oplus \quad & \textstyle\bigoplus_{\langle p,\gamma \rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\gamma'' \rangle} \quad a \otimes \Big(\bigoplus_{Q'} [p', \gamma', Q'] \otimes \\ & \qquad\qquad \textstyle\bigoplus_{\bigcup_i Q_i = Q} \bigodot_{q_j \in Q'} [q_j, \gamma'', Q_i] \Big) \\ \oplus \quad & \textstyle\bigoplus_{\langle p,\gamma \rangle \overset{a}{\hookrightarrow} \{\langle p',\gamma' \rangle, \langle p'',\gamma'' \rangle\}} \quad a \otimes \bigoplus_{Q' \cup Q'' = Q} [p', \gamma', Q'] \odot [p'', \gamma'', Q''] \end{aligned} \tag{3.4}$$

Intuitively, equation (3.4) lists the traces of all runs starting from $\langle p, \gamma\rangle$ and reaching the set of configurations $\{\langle q, \varepsilon\rangle \mid q \in Q\}$. There are five different types of runs depending on the rules that are first executed:

1. if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \emptyset$, we have $\langle p, \gamma\rangle \overset{a}{\Rrightarrow} \emptyset$,
2. if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \varepsilon\rangle$, we have $\langle p, \gamma\rangle \overset{a}{\Rrightarrow} \{\langle p', \varepsilon\rangle\}$,
3. if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\rangle$, we have $\langle p, \gamma\rangle \overset{a}{\Rrightarrow} \langle p', \gamma'\rangle \xRightarrow{[p',\gamma',Q]} \{\langle q, \varepsilon\rangle \mid q \in Q\}$,
4. if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle$, we have $\langle p, \gamma\rangle \overset{a}{\Rrightarrow} \langle p', \gamma'\gamma''\rangle \xRightarrow{[p',\gamma',Q']} \{\langle q_j, \gamma''\rangle \mid q_j \in Q'\}$, for all $Q' \subseteq P$; moreover, for all $Q_1, \ldots, Q_n \subseteq Q$ such that $\bigcup_{i=1}^{n} Q_i = Q$, we have $\langle q_j, \gamma''\rangle \xRightarrow{[q_j,\gamma'',Q_i]} \{\langle q, \varepsilon\rangle \mid q \in Q_i\}$ for each $q_j \in Q'$, and
5. if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \{\langle p', \gamma'\rangle, \langle p'', \gamma''\rangle\}$, we have $\langle p, \gamma\rangle \overset{a}{\Rrightarrow} \{\langle p', \gamma'\rangle, \langle p'', \gamma''\rangle\}$; moreover, for all $Q', Q'' \subseteq Q$ such that $Q' \cup Q'' = Q$, we have
$$\begin{array}{rcl} \langle p', \gamma'\rangle & \xRightarrow{[p',\gamma',Q']} & \{\langle q, \varepsilon\rangle \mid q \in Q'\} \text{ and} \\ \langle p'', \gamma''\rangle & \xRightarrow{[p'',\gamma'',Q'']} & \{\langle q, \varepsilon\rangle \mid q \in Q''\} \ . \end{array}$$

Equation (3.4) combines the traces of all possible runs.

Theorem 3.7 *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted alternating pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$. The least fixpoint of the system of equations (3.4) always associates $[p, \gamma, Q]$ to $T(\langle p, \gamma\rangle, \{\langle q, \varepsilon\rangle \mid q \in Q\})$, for all $p \in P$, $\gamma \in \Gamma$, and $Q \subseteq P$.*

Proof: Let $\langle Q', \emptyset\rangle$ denote $\{\langle q, \varepsilon\rangle \mid q \in Q'\}$, for any $Q' \subseteq P$. The proof is analogous to the proof of Theorem 3.6. Let μ be the least fixed point. We write $[p, \gamma, Q]_\mu$ to denote the component of μ which corresponds to the variable $[p, \gamma, Q]$.

It is easy to see that the tuple of all $T(\langle p, \gamma\rangle, \langle Q, \varepsilon\rangle)$ forms a solution of the equation system. This is done by partitioning all possible runs into disjoint subsets as discussed above. Thus, $[p, \gamma, Q]_\mu \sqsubseteq T(\langle p, \gamma\rangle, \langle Q, \varepsilon\rangle)$ for all $p \in P$, $\gamma \in \Gamma$, and $Q \subseteq P$.

To prove that $T(\langle p, \gamma\rangle, \langle Q, \varepsilon\rangle) \sqsubseteq [p, \gamma, Q]_\mu$, we first define the relation $\Rightarrow_{\leq k}$ to be the reachability relation involving only runs of lengths less than or equal to k. Formally, $\Rightarrow_{\leq} \subseteq (P \times \Gamma^*) \times D \times \mathbb{N} \times 2^{P \times \Gamma^*}$ is the smallest relation such that

- $c \overset{1}{\Rightarrow}_{\leq 0} \{c\}$, for all $c \in P \times \Gamma^*$,
- if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \{\langle p_1, w_1 \rangle, \ldots, \langle p_n, w_n \rangle\}$ and $\langle p_i, w_i w \rangle \overset{b_i}{\Rightarrow}_{\leq k} C_i$ for some $w \in \Gamma^*$, $b_i \in D$, and $C_i \subseteq P \times \Gamma^*$, for each $i \in [n]$, then
$$\langle p, \gamma w \rangle \xRightarrow{a \otimes \bigodot_{i=1}^{n} b_i}_{\leq k+1} \bigcup_{i=1}^{n} C_i \ .$$

We correspondingly define the traces for each $k \in \mathbb{N}$:

$$T_{\leq k}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) = \bigoplus \{a \mid \langle p, \gamma \rangle \overset{a}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} \ .$$

Clearly, $T(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) = \bigcup_{k \geq 0} T_{\leq k}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle))$. Therefore, showing that $T(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) \sqsubseteq [p, \gamma, Q]_\mu$ boils down to proving that the bound holds for traces of all lengths, i.e. $T_{\leq k}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) \sqsubseteq [p, \gamma, Q]_\mu$, for each $k \in \mathbb{N}$. We proceed by induction on k.

The base case, where $k = 0$, follows immediately. From the definitions of $T_{\leq k+1}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle)$ and $[p, \gamma, Q]_\mu$ we have

$$\begin{aligned}
&T_{\leq k+1}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) = \\
&\qquad \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \emptyset \ \wedge \ Q = \emptyset} && a \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \varepsilon \rangle \ \wedge \ Q = \{p'\}} && a \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma' \rangle} && a \otimes \bigoplus \{b \mid \langle p', \gamma' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma' \gamma'' \rangle} && a \otimes \bigoplus \{b \mid \langle p', \gamma' \gamma'' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \{\langle p', \gamma' \rangle, \langle p'', \gamma'' \rangle\}} && a \otimes \bigoplus \{b' \odot b'' \mid \langle p', \gamma' \rangle \overset{b'}{\Rightarrow}_{\leq k} \langle Q', \varepsilon \rangle \text{ and} \\
& && \qquad \langle p'', \gamma'' \rangle \overset{b''}{\Rightarrow}_{\leq k} \langle Q'', \varepsilon \rangle \text{ s.t. } Q' \cup Q'' = Q\}
\end{aligned}$$

and

$$\begin{aligned}
&[p, \gamma, Q]_\mu = \\
&\qquad \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \emptyset \ \wedge \ Q = \emptyset} && a \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \varepsilon \rangle \ \wedge \ Q = \{p'\}} && a \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma' \rangle} && a \otimes [p', \gamma', Q]_\mu \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', \gamma' \gamma'' \rangle} && a \otimes \Big(\bigoplus\nolimits_{Q'} [p', \gamma', Q']_\mu \otimes \\
& && \qquad \bigoplus\nolimits_{\bigcup_i Q_i = Q} \bigodot\nolimits_{q_j \in Q'} [q_j, \gamma'', Q_i]_\mu \Big) \\
&\quad \oplus \bigoplus\nolimits_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \{\langle p', \gamma' \rangle, \langle p'', \gamma'' \rangle\}} && a \otimes \bigoplus\nolimits_{Q' \cup Q'' = Q} [p', \gamma', Q']_\mu \odot [p'', \gamma'', Q'']_\mu
\end{aligned}$$

Since

$$\bigoplus\{b \mid \langle p', \gamma' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} = T_{\leq k}(\langle p', \gamma' \rangle, \langle Q, \varepsilon \rangle) \ ,$$

by the induction hypothesis we have

$$\bigoplus\{b \mid \langle p', \gamma' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} \sqsubseteq [p', \gamma', Q]_\mu \ .$$

Furthermore, from the definition of traces and the induction hypothesis

$$\begin{aligned}
&\bigoplus\{b \mid \langle p', \gamma'\gamma'' \rangle \overset{b}{\Rightarrow}_{\leq k} \langle Q, \varepsilon \rangle\} \\
&\quad\sqsubseteq \quad \bigoplus\nolimits_{Q'} T_{\leq k}(\langle p', \gamma' \rangle, \langle Q', \varepsilon \rangle) \otimes \bigoplus\nolimits_{\bigcup_i Q_i = Q} \bigodot\nolimits_{q_j \in Q'} T_{\leq k}(\langle q_j, \gamma'' \rangle, \langle Q_i, \varepsilon \rangle) \\
&\quad\sqsubseteq \quad \bigoplus\nolimits_{Q'} [p', \gamma', Q']_\mu \otimes \bigoplus\nolimits_{\bigcup_i Q_i = Q} \bigodot\nolimits_{q_j \in Q'} [q_j, \gamma'', Q_i]_\mu \ .
\end{aligned}$$

Similarly,

$$\begin{aligned}
&\bigoplus\{b' \odot b'' \mid \langle p', \gamma' \rangle \overset{b'}{\Rightarrow}_{\leq k} \langle Q', \varepsilon \rangle \wedge \langle p'', \gamma'' \rangle \overset{b''}{\Rightarrow}_{\leq k} \langle Q'', \varepsilon \rangle \text{ s.t. } Q' \cup Q'' = Q\} \\
&\quad\sqsubseteq \quad \bigoplus\nolimits_{Q' \cup Q'' = Q} T_{\leq k}(\langle p', \gamma' \rangle, \langle Q', \varepsilon \rangle) \odot T_{\leq k}(\langle p'', \gamma' \rangle, \langle Q'', \varepsilon \rangle) \\
&\quad\sqsubseteq \quad \bigoplus\nolimits_{Q' \cup Q'' = Q} [p', \gamma', Q']_\mu \odot [p'', \gamma'', Q'']_\mu \ .
\end{aligned}$$

Consequently, we conclude that $T_{\leq k+1}(\langle p, \gamma \rangle, \langle Q, \varepsilon \rangle) \sqsubseteq [p, \gamma, Q]_\mu$. $\square$

A special case

We reconsider the special case discussed in Section 3.1.2, in which computational forks can only occur when the stack content is reduced to a single symbol. Recall that a simple alternating pushdown system is a tuple $\mathcal{P} = (P, \Gamma, \Xi, \Delta)$, where $\Xi \subseteq \Gamma$, with the following properties: (i) if $\langle p, \perp \rangle$, where $p \in P$ and $\perp \in \Xi$, is the root of an computational tree, then all nodes are of the form $\langle q, w\perp' \rangle$, where $q \in P$, $w \in (\Gamma \setminus \Xi)^*$, and $\perp' \in \Xi$; (ii) if a configuration c of a computational tree has more than one child, then $c = \langle p, \perp \rangle$, for some $p \in P$ and $\perp \in \Xi$. It follows that if a configuration $\langle p, w \rangle$, where $p \in P$ and $w \in (\Gamma \setminus \Xi)^*$, is the root of a tree, then every configuration of the tree has at most one child, and so the tree has a unique leaf. We exploit this fact in our solution.

We show that it is possible to characterize the sets of traces of such models by equation systems of polynomial sizes; contrary to general alternating pushdown systems, where the associated equation systems are of exponential sizes. The variables are of the form $[p, \perp, q]$ or $[p, \gamma, q]$, where $p, q \in P$,

$\perp \in \Xi$ and $\gamma \in \Gamma \setminus \Xi$. The variable $[p, \perp, q]$ represents the traces of all runs starting at $\langle p, \perp \rangle$ and eventually reaching a tree where all leaves are labeled with $\langle q, \perp_1 \rangle$ for some $\perp_1 \in \Xi$. The variable $[p, \gamma, q]$ represents the traces of all runs starting at $\langle p, \gamma \rangle$ and reaching a tree whose unique leaf (from the fact above) is labeled with $\langle q, \varepsilon \rangle$.

Similar to the general case, we assume without loss of generality two restrictions on rules: the restriction (R1) in Section 3.1.2 for non-alternating rules and for every alternating rule $\langle p, \gamma \rangle \hookrightarrow R$ in Δ:

(R2′) $|R| = 2$ and $\forall \langle p', w' \perp' \rangle \in R : |w'| \leq 1$.

The restriction (R2′) is slightly modified from (R2) in Section 3.1.2 to incorporate the fact that an alternating rule always involves a symbol from Ξ at the bottom, with an optional symbol from $\Gamma \setminus \Xi$ on the top.

Again, we show that the set of traces of all runs running from given configurations to other configurations can be characterized as the least fixpoint of a system of equations. For all $p, q \in P$ and $\gamma \in \Gamma$, we define the variables as the triples $[p, \gamma, q]$, and the set of equations as follows:

$$\begin{aligned}
[p,\gamma,q] = & \\
& \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle q,\varepsilon\rangle} && a \\
\oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\rangle} && a \otimes [p', \gamma', q] \\
\oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',\gamma'\gamma''\rangle} && a \otimes \Big(\bigoplus_{p'' \in P} [p', \gamma', p''] \otimes [p'', \gamma'', q] \Big) \\
\oplus & \bigoplus_{\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \{\langle p_i,\gamma_i \perp_i\rangle \mid i \in [2]\}} && a \otimes \Big(\bigoplus_{q_1,q_2 \in P} \bigodot_{i=1}^{2} [p_i, \gamma_i, q_i] \otimes [q_i, \perp_i, q] \Big) \ .
\end{aligned} \tag{3.5}$$

Moreover, we set $[p, \perp, p] = 1$, for all $\perp \in \Xi$; and $[p, \varepsilon, q] = 1$ if $p = q$, and 0 otherwise.

Intuitively, given $[p, \gamma, q]$, where $p, q \in P$ and $\gamma \in \Gamma \setminus \Xi$, the first three parts of equation (3.5), which is exactly equation (3.3), list the traces of all runs starting from $\langle p, \gamma \rangle$ and reaching $\langle q, \varepsilon \rangle$. Notice that when $\gamma \in \Gamma \setminus \Xi$ theses traces involve only non-alternating rules. The last part of the equation deals with alternating rules. Given $[p, \perp, q]$, where $p, q \in P$ and $\perp \in \Xi$, it lists the traces of all runs starting from $\langle p, \perp \rangle$ and reaching $\langle q, \perp' \rangle$ for any $\perp' \in \Xi$. The idea is that $\langle p, \perp \rangle \overset{a}{\hookrightarrow} \{\langle p_1, \gamma_1 \perp_1 \rangle, \langle p_2, \gamma_2 \perp_2 \rangle\}$, corresponds to the run

$$\langle p, \perp \rangle \overset{a}{\Rightarrow} \{\langle p_1, \gamma_1 \perp_1 \rangle, \langle p_2, \gamma_2 \perp_2 \rangle\} \ .$$

Moreover, for any $q_1, q_2 \in P$, we have

$$\langle p_i, \gamma_i \bot_i \rangle \xRightarrow{[p_i,\gamma_i,q_i]} \langle q_i, \bot_i \rangle$$

for all $i \in [2]$, and $[q_i, \bot_i, q]$ contains all traces from $\langle q_i, \bot_i \rangle$ to $\langle q, \bot' \rangle$ for any $\bot' \in \Xi$. Equation (3.5) combines the traces of all possible runs.

Theorem 3.8 *Let $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted alternating pushdown system, where $\mathcal{P} = (P, \Gamma, \Xi, \Delta)$ is a simple alternating pushdown system and $\mathcal{S} = (D, \oplus, \otimes, \odot, 0, 1, 1')$. The least fixpoint of the system of equations (3.5) always associates*

1. *$[p, \gamma, q]$ to $T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$, for all $p, q \in P$ and $\gamma \in \Gamma \setminus \Xi$; and*
2. *$[p, \bot, q]$ to $\bigoplus_{\bot' \in \Xi} T(\langle p, \bot \rangle, \langle q, \bot' \rangle)$, for all $p, q \in P$ and $\bot \in \Xi$.*

Proof: Let μ be the least fixed point. We write $[p, \gamma, q]_\mu$ (resp. $[p, \bot, q]_\mu$) to denote the component of μ which corresponds to the variable $[p, \gamma, q]$ (resp. $[p, \bot, q]$).

We proceed similarly to the previous proofs. It is easy to see that the tuple of all $T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$ and $\bigoplus_{\bot' \in \Xi} T(\langle p, \bot \rangle, \langle q, \bot' \rangle)$ forms a solution of the equation system. This is done by partitioning all possible runs into disjoint subsets as discussed above. Thus, $[p, \gamma, q]_\mu \sqsubseteq T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle)$ and $[p, \bot, q]_\mu \sqsubseteq \bigoplus_{\bot' \in \Xi} T(\langle p, \bot \rangle, \langle q, \bot' \rangle)$ for all $p, q \in P$, $\gamma \in \Gamma \setminus \Xi$, and $\bot \in \Xi$.

To prove that $T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle) \sqsubseteq [p, \gamma, q]_\mu$ and $\bigoplus_{\bot' \in \Xi} T(\langle p, \bot \rangle, \langle q, \bot' \rangle) \sqsubseteq [p, \bot, q]_\mu$, we first define the relation $\Rightarrow_{\leq k}$ to be the reachability relation involving only runs of lengths less than or equal to k. Formally, $\Rightarrow_{\leq} \subseteq (P \times \Gamma^*) \times D \times \mathbb{N} \times (P \times \Gamma^*)$ is the smallest relation such that

- $c \overset{1}{\Rightarrow}_{\leq 0} c$, for all $c \in P \times \Gamma^*$,
- if $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$ and $\langle p', ww' \rangle \overset{b}{\Rightarrow}_{\leq k} c$ for some $w' \in \Gamma^*$, $b \in D$, and $c \in P \times \Gamma^*$, then $\langle p, \gamma w' \rangle \xRightarrow{a \otimes b}_{\leq k+1} c$.
- if $\langle p, \bot \rangle \overset{a}{\hookrightarrow} \{\langle p_1, \gamma_1 \bot_1 \rangle, \langle p_2, \gamma_2 \bot_2 \rangle\}$ and $\langle p_i, \gamma_i \bot_i \rangle \overset{b_i}{\Rightarrow}_{\leq k} c$ for some $b_i \in D$ and $c \in P \times \Gamma$, for each $i \in [2]$, then $\langle p, \bot \rangle \xRightarrow{a \otimes (b_1 \odot b_2)}_{\leq k+1} c$.

We correspondingly define the traces for each $k \in \mathbb{N}$:

$$T_{\leq k}(c, c') = \bigoplus \{a \mid c \overset{a}{\Rightarrow}_{\leq k} c'\} \ .$$

Clearly, $T(c, c') = \bigcup_{k \geq 0} T_{\leq k}(c, c')$. Therefore, showing that

$$T(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle) \sqsubseteq [p, \gamma, q]_\mu \text{ and } \bigoplus_{\perp' \in \Xi} T(\langle p, \perp \rangle, \langle q, \perp' \rangle) \sqsubseteq [p, \gamma, q]_\mu$$

boils down to proving

$$T_{\leq k}(\langle p, \gamma \rangle, \langle q, \varepsilon \rangle) \sqsubseteq [p, \gamma, q]_\mu \text{ and } \bigoplus_{\perp' \in \Xi} T_{\leq k}(\langle p, \perp \rangle, \langle q, \perp' \rangle) \sqsubseteq [p, \perp, q]_\mu \ ,$$

for each $k \in \mathbb{N}$. The proof for the non-alternating part can be found in the proof of Theorem 3.6, hence omitted here. We prove the alternating part by induction on k.

The base case, where $k = 0$, follows immediately. From the definitions of $\bigoplus_{\perp' \in \Xi} T_{\leq k+1}(\langle p, \perp \rangle, \langle q, \perp' \rangle)$ and $[p, \perp, q]_\mu$ we have

$$\bigoplus_{\perp' \in \Xi} T_{\leq k+1}(\langle p, \perp \rangle, \langle q, \perp' \rangle) \sqsubseteq \bigoplus_{\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \{\langle p_i, \gamma_i \perp_i \rangle | i \in [2]\}} a \otimes b \ ,$$

where

$$b = \bigoplus_{q_1, q_2 \in P} \bigodot_{i=1}^{2} T_{\leq k}(\langle p_i, \gamma_i \rangle, \langle q_i, \varepsilon \rangle) \otimes \left(\bigoplus_{\perp' \in \Xi} T_{\leq k}(\langle q_i, \perp_i \rangle, \langle q, \perp' \rangle) \right) \ ,$$

and

$$[p, \perp, q]_\mu = \bigoplus_{\langle p, \perp \rangle \overset{a}{\hookrightarrow} \{\langle p_i, \gamma_i \perp_i \rangle | i \in [2]\}} a \otimes \bigoplus_{q_1, q_2 \in P} \bigodot_{i=1}^{2} [p_i, \gamma_i, q]_\mu \otimes [q_i, \perp_i, q]_\mu \ .$$

By the induction hypothesis we have

$$T_{\leq k}(\langle p_i, \gamma_i \rangle, \langle q_i, \varepsilon \rangle) \sqsubseteq [p_i, \gamma_i, q_i]_\mu$$

and

$$\bigoplus_{\perp' \in \Xi} T_{\leq k}(\langle q_i, \perp_i \rangle, \langle q, \perp' \rangle) \sqsubseteq [q_i, \perp_i, q]_\mu \ ,$$

for each $i \in \{1, 2\}$. Consequently, we conclude that

$$\bigoplus_{\perp' \in \Xi} T_{\leq k+1}(\langle p, \perp \rangle, \langle q, \perp' \rangle) \sqsubseteq [p, \perp, q]_\mu \ .$$

□

Chapter 4

Application to Java testing

In this chapter we describe an application of the reachability analysis on pushdown models to the area of Java testing. Given a Java program to be tested, the program is first compiled into so-called class files containing Java *bytecode*—the machine language of the Java virtual machine. We then construct a pushdown model from the class files such that the reachability analysis simulates the behavior of the bytecode as if it were executed by the Java virtual machine. In contrast to the traditional testing where test cases are executed one after another by the Java virtual machine, one can think of the reachability analysis as a single execution of all possible parameter values within given ranges. Therefore, we are able to find out not only all errors in the ranges, but also parts of the code that are not reachable by any values. There are two obvious advantages over the traditional testing. First, the reachability analysis always terminates, which is not always the case in the traditional testing where infinite loops are possible. Also, it is possible to test wider ranges of parameter values with the reachability analysis, thus increasing more confident about the correctness of the program under test.

To make the testing possible, we need to solve at least two major problems: (i) Given a class file, how can one construct a pushdown model that simulates the behavior of the Java virtual machine? (ii) How can one define weights of pushdown models such that the reachability analyses can be performed efficiently? We answer the first problem by means of the first two sections. The first section introduces the Java virtual machine and the basics of Java bytecode. The second section explains a translation of bytecode instructions into pushdown models. The third section focuses on the second problem, where we need to represent relations as weights. We discuss several issues that arise when applying the reachability algorithm from Section 3.1. An algorithm that extracts concrete parameter values at a given (reachable) program point is also proposed at the end of the chapter.

4.1 Java virtual machine

This section introduces the concept of the Java virtual machine (JVM) needed for understanding the rest of the chapter. We do not list all aspects of the JVM in this thesis. Readers are referred to [41] for a thorough explanation of the JVM specification. Some texts in this section are taken either directly or with minor modifications from [41].

The JVM is an abstract computing machine that runs compiled Java programs. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. The JVM is a software available on many hardware platforms and operating systems, resulting in the portability of Java programs.

The JVM operates on two kinds of data types: *primitive types* and *reference types*. The primitive types consist of the numeric types, the `boolean` type, and the `returnAddress` type. The numeric types are `bytes` (8 bits), `short` (16 bits), `char` (16 bits), `int` (32 bits), `float` (32 bits), `long` (64 bits), and `double` (64 bits). The `boolean` type supports the truth values `true` and `false`. The value of the `returnAddress` type are pointers to opcodes of the JVM instructions. Object references are of type `reference`. An object can be either a (dynamically created) class instance or an array. One can think of a reference as a pointer to an object.

The JVM maintains various runtime information during execution of a program. The *method area* stores bytecode instructions and other class-related information. The *heap* is where objects, i.e. class instances and arrays, are allocated. Objects can never be explicitly allocated. Instead, the memory occupied by objects is reclaimed by an automatic storage management system when they are no longer referenced—a process known as a garbage collection. For each thread, the *pc* (program counter) register points to the currently executing bytecode instruction in the method area. After an instruction is executed, the pc register points to the next instruction, either the one that immediately follows or a jump specified by the instruction. Each thread also keeps track of a stack of *frames* for method invocations and returns. A new frame is created when a method is invoked. Then, it becomes the current frame where the control transfers to. A frame is destroyed when its method invocation completes, either a normal completion or an uncaught exception is thrown. Upon return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Each frame possesses an array of *local variables* and an *operand stack*. A local variable can hold a value of all types that require up to 32 bits. A consecutive pair of local variables is needed to hold a value of type `long` or `double`. Local variables are addressed by indexing, starting from index zero. Values of type `long` or `double` are addressed using the lesser indices. The length of the local variable array of a frame is determined at compile time from its associated method, and stored in the corresponding class file.

Although the Java virtual machine has no registers for storing temporary values, it maintains something equivalent called operand stack. The operand stack is initially empty when the frame that contains it is created. The operand stack can be loaded with constants or values from local variables or fields. Many instructions pop operands from the stack, operate on them, and push the result back. Operand stacks are also used to prepare parameters for method calls and receive method return values. The maximum depth of the operand stack of a frame is determined at compile time from its associated method, and stored in the corresponding class file.

4.1.1 Java bytecode basics

Java bytecode is the form of instructions that the JVM executes. Each instruction consists of a one-byte opcode immediately followed by zero or more operands. The numbers and types of operands are determined by the opcodes. As in the typical assembly language style, instructions are usually represented by their mnemonics followed by their operand values. In the following, we always refer to the instructions by their mnemonics.

Each (non-abstract) method has a list of instructions, which will be executed by the JVM starting from the first instruction when the method is active. The first instruction always has *offset* 0. The offsets of the next instructions are equal to the offsets of the previous instructions plus the sizes of the previous instructions (in bytes). Figure 4.1 shows a Java method, which simply stays in the loop 10 times without doing anything else, together with its bytecode instructions. Let us consider the meaning of each bytecode instruction in this example. As discussed earlier, the JVM has no register for

Java source:

```
static void local() {
    int i;
    for (i = 0; i < 10; i++)
        ;
}
```

Bytecode instructions:

```
local()V
 0: iconst_0
 1: istore_0
 2: goto 8
 5: iinc 0 by 1
 8: iload_0
 9: bipush 10
11: if_icmplt 5
14: return
```

Figure 4.1: A Java method that loops 10 times

storing temporary values. Everything must be pushed onto an operand stack before it can be used in a calculation. Bytecode instructions therefore operate primarily on the operand stack. In the example, the execution starts with the instruction `iconst_0` by pushing the constant zero of type integer onto the operand stack. With `istore_0`, it pops the integer value from the operand stack, which is the zero, and stores it into the local variable at index 0. These two instructions correspond to the statement `i = 0`. Then, the execution unconditionally branches to the instruction at offset 8 because of the instruction `goto 8`. There, `iload_0` reads the current integer value from the local variable at index 0, and pushes it onto the operand stack. The instruction `bipush 10` converts the value 10 of type byte to type integer, and pushes it onto the operand stack. With `if_icmplt 5` (if integer comparison less than), two values, which must be of type integer, are popped from the operand stack, and the execution branches to offset 5 if the second topmost value is less than the topmost value, otherwise it continues to the next instruction. The instructions at offsets 8–11 therefore correspond to the comparison `i < 10` in the for-loop. At offset 5, `iinc 0 by 1` increases the local variable at index 0 by 1, corresponding to the statement `i++`. The loop repeats by checking the value of `i` again starting at offset 8. The method `return`s when the execution reaches offset 14.

Similar to `iload` and `istore`, the JVM contains two instructions for reading from and writing to static fields: `getstatic` and `putstatic`. For instance, the instruction `getstatic C.x` pushes the value of the static field `x` that belongs to the class `C` onto the operand stack. The instruction `putstatic C.x`, on the other hand, pops the value from the operand stack and stores it in `x`.

Passing parameters and return values

When invoking a method, the JVM uses the operand stack of the caller frame and the local variables of the new frame to pass parameters. The JVM internally pops the values from the operand stack and stores them into the local variables such that the first parameter is the depth-most value on the operand stack. There are two types of method invocations in Java: class method invocation and instance method invocation. In the case of class method invocation, parameters are passed in consecutive local variables starting from index 0. In the case of instance method invocation, local variable at index 0 is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Parameters are passed in consecutive local variables starting from index 1.

When a method returns a value, the value is pushed onto the operand stack of the frame of the invoker. Consider an example in Figure 4.2, where two static methods are shown. The method `tt` calls the method `sub` with 3 and 2 as the arguments. This is done it three steps. First, the integer constant 3 is pushed, then the constant 2 is pushed before calling the method `sub`. The instruction `invokestatic` pops the argument values from the operand stack, and constructs a new frame for the method `sub`. In the method `sub` the values 3 and 2 are stored in the local variables 0 and 1, respectively. The instructions `iload_0` and `iload_1` load these two values onto the operand stack. The instruction `isub` pops them, subtracts, and pushes the result back. With `ireturn`, the result is popped, the frame of the method `sub` is discarded, and the popped value is pushed back onto the operand stack of the frame of `tt`. This value is returned again by `ireturn` in `tt`.

Arrays

The JVM has a distinct set of instructions for manipulating arrays. We discuss some of them by considering the example in Figure 4.3. The instruction `newarray int` creates a new array of type integer by first popping an integer value from the operand stack, and then allocating the array of length specified by the popped value from the heap. A reference to the array is

Java source:

```
static int sub(int i, int j) {
    return i - j;
}
```

Bytecode instructions:

```
sub(II)I
 0: iload_0
 1: iload_1
 2: isub
 3: ireturn
```

```
static int tt() {
    return sub(3, 2);
}
```

```
tt()V
 0: iconst_3
 1: iconst_2
 2: invokestatic sub(II)I
 5: ireturn
```

Figure 4.2: An example that illustrates parameters passing

pushed onto the operand stack. Next, the reference is stored in the local variable 0 by `astore_0`. Therefore, the first three instructions correspond to the statement `int[] a = new int[2];`. The instruction `aload_0` loads the array reference back onto the operand stack. Notice that `astore_0` and `aload_0` are the same as `istore_0` and `iload_0`, except that the first two require values of type `reference`, instead of type `int`.

The instruction `arraylength` pops an array reference from the operand stack, and pushes the length of the array specified by the reference onto the stack. The instruction `iastore` pops a value of type `int`, an array index, and an array reference, exactly in this order. Then, the value is stored in the array element specified by the index. In the example, the instructions at offsets 4–8 correspond to the statement `a[0] = a.length;`. The instruction `iaload` pops an array index and an array reference, and pushes the array element specified by the index onto the operand stack.

Class instances

In Java, a class instance can be created by using the `new` keyword, see Figure 4.4 for an example. The bytecode instruction `new` allocates a memory for a new instance from the heap, initializes the instance variables to their default values, and pushes a reference to the instance onto the operand stack.

Java source:

```
static int ar() {
    int[] a = new int[2];
    a[0] = a.length;
    return a[1];
}
```

Bytecode instructions:

```
ar()I
 0: iconst_2
 1: newarray int
 3: astore_0
 4: aload_0
 5: iconst_0
 6: aload_0
 7: arraylength
 8: iastore
 9: aload_0
10: iconst_1
11: iaload
12: ireturn
```

Figure 4.3: An example that manipulates an array

The instruction `dup` duplicates the top element of the operand stack The instruction `invokespecial` is used here for invoking the instance initialization method, i.e. the constructor. Notice that the name of constructors is always the compiler-supplied `<init>`. The instruction behaves just like an instance method invocation, and therefore when executed, an object reference and method arguments are popped from the operand stack. Then, a new frame is created on the stack, and the local variables of the new frame are initialized with the values of the object reference and the arguments, with the reference in local variable 0, the first argument in local variable 1, and so on. In the example, the constructor takes no arguments, so only the object reference is passed to the new frame. After the call, there is one reference left on the operand stack. This reference is subsequently popped, and stored in local variable 0 by `astore_0`. Thus, the instructions at offsets 0–7 correspond to the statement `B o = new B();`.

The instruction `instanceof A` pops a reference from the operand stack, and determines whether its object is an instance of class `A` or a subclass of `A`. If yes, an integer 1 is pushed onto the operand stack, otherwise an integer 0 is pushed. The instruction `ifeq 20` pops an integer from the operand stack, and branches to the instruction at offset 20 if the value is equal to zero.

Java source:

```
class A {
    int x;
}

class B extends A {
    static void in() {
        B o = new B();
        if(o instanceof A) {
            o.x = 1;
        }
    }
}
```

Bytecode instructions:

```
B.in()V
 0: new B
 3: dup
 4: invokespecial B.<init>()V
 7: astore_0
 8: aload_0
 9: instanceof A
12: ifeq 20
15: aload_0
16: iconst_1
17: putfield B.x
20: return
```

Figure 4.4: An example that manipulates a class instance

Otherwise, the execution continues to the next instruction. In the example, the execution does not branch, since `B` is a subclass of `A`. The instruction `putfield` pops a value and a reference from the operand stack, and stores the value to a field of the instance specified by the reference.

Exceptions

In Java programming language, an exception can be thrown using the `throw` keyword, which results in an immediate transfer of control to the nearest enclosing `catch` clause of a `try` statement that handles the exception. Each `catch` clause is represented by an *exception handler*. An exception handler h specifies (i) the range r_h of instruction offsets for which it is active, (ii) the exception type t_h that it is able to handle, and (iii) the code location l_h that is to handle that exception. An exception matches an exception handler h if the offset of the instruction that causes the exception is in the range r_h and the exception type is the same class as or a subclass of t_h. When an exception is thrown, the JVM searches for a matching exception handler in the current method. If a matching exception handler h is found, the execution branches to the exception handling code specified by l_h.

If a matching exception cannot be found in the current method, the current method invocation completes abruptly. The operand stack and local variables of the current frame are discarded, the frame is popped, and the next frame on the stack—the frame of the invoker method—is reinstated. The exception is again thrown in the context of the invoker's frame, and continue popping the stack of frames until an exception handler is found. If no exception handler can be found in the last frame of the stack, the execution of the corresponding thread terminates.

For each method, the exception handlers are stored in a table within the class file. When an exception is thrown, the JVM sequentially searches for a matching exception handler in the table, starting from the first entry. Figure 4.5 illustrates a small method together with its bytecode instructions and exception table. The instructions at offsets 0–4 create and initialize an object of type `RuntimeException`, and at affset 7 the object is thrown by `athrow`. The only exception handler specified in the table handles `Exception` that can be thrown from instructions at the offsets 0 (inclusive) to 8 (exclusive). Therefore, the exception thrown from offset 7 always matches the exception handler, and the execution subsequently branches to offset 8. The exception is thrown again at offset 10, however no matching exception handler can be found. The exception then propagates further to the invoker's frame.

Multithreading

There are two ways to create a new thread in Java. The first approach is to declare a class to be a subclass of `java.lang.Thread`. An instance of this subclass can start the thread directly. The other approach is to declare a class that implements the interface `java.lang.Runnale`. After that, an instance of this class can be passed as an argument when creating an instance of type `Thread`, and then started. Either way, the `run` method, which contains the code of the new thread, must be implemented.

Figure 4.6 gives an example of the second approach. At the beginning of `f`, the instruction `new Thread` creates a new object of type `Thread`, and pushes a reference to the object onto the operand stack. The instruction `dup` duplicates the top element of the operand stack. At offset 4, an object of type `C$1` is allocated. Class `C$1` is an inner class of `C` which implements the interface `Runnable`. `C$1` specifies the method `run` which will be executed when the thread starts.

Two initialization methods are called at offsets 8 and 11 for `C$1` and `Thread`, respectively. A reference to `C$1` (resp. to `Thread`) is passed as the first argument when initializing the `C$1` (resp. `Thread`) object. However,

Java source:

```
static void ex() throws Exception {
    try { throw new RuntimeException(); }
    catch (Exception e) { throw e; }
}
```

Bytecode instructions:

```
ex()V
 0: new java/lang/RuntimeException
 3: dup
 4: invokespecial java/lang/RuntimeException.<init>()V
 7: athrow
 8: astore_0
 9: aload_0
10: athrow
```

Exception table:

From	To	Target	Type
0	8	8	`java/lang/Exception`

Figure 4.5: An example that throws an exception

for Thread, a reference to C$1 is also passed as the second argument, which will be stored as its field when initializing the object (codes not shown). The thread is started at offset 14 by a call to start. Internally, the method start starts the thread by a call to start0 (offset 29).

Synchronization

Each object has a monitor associated with it. When a thread executes the bytecode instruction monitorenter, an object reference is popped from the operand stack, and the thread gains the ownership of the monitor associated with the reference, if the monitor is not owned by any thread. If the monitor is currently owned by another thread, the current thread waits until the monitor is released, then tries again to gain the ownership. If the current thread already owns the monitor, it increments a counter in the monitor indicating the number of times the thread has entered the monitor.

The bytecode instruction monitorexit pops an object reference, decrements the counter of the monitor associated with the reference. If the value of the counter becomes zero, the current thread releases the monitor. In this case, other threads that are waiting for the monitor are allowed to try to acquire it.

4.1.2 Instruction set

Most instructions encode information on the types of variables they operate. For instance, iload instruction loads the content of a local variable, which must be an integer, onto the operand stack. The lload, fload, and dload do the same with a long, float, and double value, respectively. Notice that letters prefixing the mnemonics indicate types. There are eight of them: a for reference, b for byte, c for char, d for double, f for float, i for int, l for long, and s for short. Some instructions, whose types are unambiguous, do not have a type letter in their mnemonics. For instance, arraylength always operates on an object that is an array. Some instructions, such as goto—an unconditional jump, do not operate on typed operands.

Note, however, that not all instructions are available for every data type. For instance, there is a load instruction of type int, i.e. iload, but there is no load instruction of type byte. In fact, most instructions do not support the types byte, char, and short. None of them support the boolean type. Instead, they are either sign-extended or zero-extended to type int. Oper-

Java source:

```
class C {
    static void f() {
        new Thread(new Runnable() {
            public void run() {
                // New thread works
        }}).start();
        // Main thread works
    }
}
```

Bytecode instructions:

```
C.f()V
 0: new Thread
 3: dup
 4: new C$1
 7: dup
 8: invokespecial C$1.<init>()V
11: invokespecial Thread.<init>(Ljava/lang/Runnable;)V
14: invokevirtual Thread.start()V
17: ...
 e: return

java/lang/Thread.start()V
 0: ...
28: aload_0
29: invokespecial java/lang/Thread.start0()V
32: ...
```

Figure 4.6: A example that forks a new thread

ations on values of these types are performed by instructions operating on values of type `int`.

We now summarize the instruction set. Some instructions are grouped together as they have similar behaviors. Boldfaced letters in the front indicate group names. These names will be used later in Section 4.2.3.

Load and store instructions

The following instructions transfer values from/to local variables, operand stacks, or static fields. Instructions with trailing brackets denote families of instructions, e.g. `fconst_[0,2]` denotes `fconst_0`, `fconst_1`, `fconst_2`, which push constants 0, 1, 2, respectively, of type `float`. Such instructions are specializations of generic instructions, e.g. `fconst`, that take one operand. For the specialized instructions, the operand is implicit and does not need to be stored or fetched. The semantics are otherwise the same, e.g. `fconst_0` has the same meaning as `fconst` with the operand `0`.

Push Push a constant onto the operand stack: `bipush`, `sipush`, `ldc`, `ldc_w`, `ldc2_w`, `aconst_null`, `iconst_m1`, `iconst_[0,5]`, `lconst_[0,1]`, `f-const_[0,2]`, `dconst_[0,1]`.

Load Load a local variable onto the operand stack: `iload`, `iload_[0,3]`, `lload`, `lload_[0,3]`, `fload`, `fload_[0,3]`, `dload`, `dload_[0,3]`, `aload`, `a-load_[0,3]`.

Store Store a value from the operand stack into a local variable: `istore`, `istore_[0,3]`, `lstore`, `lstore_[0,3]`, `fstore`, `fstore_[0,3]`, `dstore`, `d-store_[0,3]`, `astore`, `astore_[0.3]`.

Globalload Load a static field onto the operand stack: `getstatic`.

Globalstore Store a value from the operand stack into a static field: `put-static`.

Arithmetic instructions

The following instructions take values from local variables or operand stacks, compute results, and put them back.

Unary Unary operation: pop a value from the operand stack, compute the result, and push it back:

- Negation: `ineg`, `lneg`, `fneg`, `dneg`.
- Type conversion: `i2b`, `i2s`, `i2l`, `i2f`, `i2d`, `l2i`, `l2f`, `l2d`, `f2i`, `f2l`, `f2d`, `d2i`, `d2l`, `d2f`.

Binary Binary operation: pop two values from the operand stack, compute the result, and push it back:

- Addition: `iadd`, `ladd`, `fadd`, `dadd`.
- Subtraction: `isub`, `lsub`, `fsub`, `dsub`.
- Multiplication: `imul`, `lmul`, `fmul`, `dmul`.
- Division: `idiv`, `ldiv`, `fdiv`, `ddiv`.
- Remainder: `irem`, `lrem`, `frem`, `drem`.
- Shift: `ishl`, `ishr`, `iushr`, `lshl`, `lshr`, `lushr`.
- Bitwise OR: `ior`, `lor`.
- Bitwise AND: `iand`, `land`.
- Bitwise exclusive OR: `ixor`, `lxor`.
- Comparison: `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl`, `lcmp`

Inc Increment a local variable: `iinc`.

Control transfer instructions

The following instructions transfer controls to instructions other than ones immediately follow.

If Pop a value from the operand stack and branch if the comparison between the value and a given value succeeds: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `tableswitch`, `lookupswitch`.

Ifcmp Pop two values from the operand stack and breach if the comparison between the two values succeeds: `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`.

Goto Unconditional branch: `goto`, `goto_w`.

Invoke Invoke a new method. Parameters are popped from the operand stack, if any: `invokevirtual`, `invokeinterface`, `invokespecial`, `invokestatic`.

Return Return from the current method. A return value is popped from the operand stack, if any: `return`, `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`.

Object manipulation instructions

The following instructions are used for manipulating objects, including arrays.

Newarray Pop an array length (more than one time if multi-dimensional), create a new array of the given length(s), and push its reference onto the operand stack: `newarray`, `anewarray`, `multianewarray`.

Arraylength Pop an array reference from the operand stack and push the length of the array specified by the reference onto the operand stack: `arraylength`.

Arrayload Pop an array index and an array reference from the operand stack, and load the array element at the index onto the operand stack: `baload`, `caload`, `saload`, `iaload`, `laload`, `faload`, `daload`, `aaload`.

Arraystore Pop a value, an array index, and an array reference from the operand stack, and store the value into the array element at the index: `bastore`, `castore`, `sastore`, `iastore`, `lastore`, `fastore`, `dastore`, `aastore`.

New Create a new class instance and push a reference to the new instance onto the operand stack: `new`.

Fieldload Pop an object reference from the operand stack and load a field of the instance specified by the reference onto the operand stack: `getfield`.

Fieldstore Pop a value and an object reference from the operand stack and store the value into a field of the instance specified by the reference: `putfield`.

Instanceof Pop an object reference from the operand stack and determine whether the instance specified by the reference is of a given type: `instanceof`, `checkcast`.

Other instructions

Pop Pop the operand stack: `pop`, `pop2`.

Dup Duplicate the operand stack: `dup`, `dup2`, `dup_x1`, `dup2_x1`, `dup_x2`, `dup2_x2`.

Swap Swap the top two operand stack values: `swap`.

Monitorenter Enter the monitor of an object: `monitorenter`.

Monitorexit Exit the monitor of an object: `monitorexit`.

Throw Throw an exception: `athrow`.

Finally Implement finally: `jsr`, `jsr_w`, `ret`.

4.2 Translator

Given a Java class file, the goal of this section is to generate a pushdown model so that the reachability analysis in Chapter 3 can be applied. We first focus only on modeling of sequential programs with pushdown systems before extending to pushdown networks for multithreaded programs.

Java source:

```
static void a() {
    b();
}

static void b() {
    while (true) {}
}
```

Bytecode instructions:

```
a()V
 0: invokestatic b
 3: return

b()V
 0: goto 0
```

Figure 4.7: An example without variables

4.2.1 Control flow modeling

Programs without variables are translated into pushdown systems of the form $(\{\cdot\}, \Gamma, \Delta)$, where the stack symbols are program points constructed from each bytecode instruction of interest. In the following, we shall use method names followed by instructions offsets as program points. Consider an example in Figure 4.7, where three bytecode instructions are involved. In this case, we define $\Gamma = \{\mathtt{a}_0, \mathtt{a}_3, \mathtt{b}_0\}$. Notice that care must be taken when two methods have the same name. To avoid this problem, our translator always attaches package names, class names, and method signatures alongside the method names to guarantee the uniqueness. For readability reasons, however, only method names are listed in our examples when there is no ambiguity.

For sequential programs, the set Δ contains rules that transform stack symbols into zero, one, or two stack symbols, i.e. there are three possible types of rules programs can be translated into. For arbitrary stack symbols γ, γ', and γ'',

1. *Normal rule* $\langle \cdot, \gamma \rangle \hookrightarrow \langle \cdot, \gamma' \rangle$ transforms program point γ to γ'.
2. *Push rule* $\langle \cdot, \gamma \rangle \hookrightarrow \langle \cdot, \gamma'\gamma'' \rangle$ calls a new method at program point γ, where γ' is the method's entry point and γ'' is the return address.
3. *Pop rule* $\langle \cdot, \gamma \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle$ returns from a method at program point γ.

Notice that these three types of rules obey the restriction imposed by the algorithm in Section 3.1.1. Therefore, the algorithm can be directly applied to the translated pushdown systems without any modifications.

Rules		Relation types
Pop rule	$\gamma_1 \hookrightarrow \varepsilon$	$G \times L \times G$
Normal rule	$\gamma_1 \hookrightarrow \gamma_2$	$G \times L \times G \times L$
Push rule	$\gamma_1 \hookrightarrow \gamma_2\gamma_3$	$G \times L \times G \times L \times L$

Table 4.1: Pushdown rules and their relation types

Since pushdown systems of programs without variables always contain only one control location, we will drop it when writing rules from now on, i.e. we write $\gamma \hookrightarrow w$ instead of $\langle \cdot, \gamma \rangle \hookrightarrow \langle \cdot, w \rangle$. Continuing with the example in Figure 4.7, we construct a rule for each bytecode instruction: $\Delta = \{\mathtt{a}_0 \hookrightarrow \mathtt{b}_0\mathtt{a}_3,\ \mathtt{a}_3 \hookrightarrow \varepsilon,\ \mathtt{b}_0 \hookrightarrow \mathtt{b}_0\}$.

4.2.2 Variable modeling

The data part of a program is modeled using weights. The weight of a rule is a relation describing the possible pairs of valuations of the variables before and after executing the rule. Notice that the resulting model effectively separates control and data parts. A weighted rule has the form:

$$\gamma \hookrightarrow w \quad R \ ,$$

where $\gamma \in \Gamma$, $w \in \Gamma^*$ and $|w| \leq 2$, and R is a relation between variables.

As usual, variables are divided into *globals* and *locals*[1]. Globals are shared among all program points; whereas each program point can own a copy of locals. Therefore, the definition of R depends on the type rules it is assigned to. Let G and L be valuation sets for globals and locals, respectively. Table 4.1 lists all possible relation types, depending on the numbers of right-hand stack symbols. Intuitively, a weight can be seen as a mechanism which updates globals and locals for each program point involved in the rule.

For instance, if the model contains a global g and a local l of type integer, then the rule

$$\gamma_1 \hookrightarrow \gamma_2\gamma_3 \quad \{(g, l, 0, l+1, l) \mid g \in G \text{ and } l \in L\} \ ,$$

encodes a procedure call from γ_1 to γ_2, with γ_3 as the return address. At the same time, g is updated to 0, x of γ_2 is greater than x of γ_1 by 1, and x of γ_3 remains unchanged.

[1]We use the term *locals* here to differentiate them from Java's *local variables*, whereas the term *variables* is more general and will be used in both contexts.

In the following, we will implicitly write relations as expressions over variables. Primes are used to distinguish between copies of variables in relations. As a convention, unprimed variables refer to the stack symbol on the left-hand side of the rule. Singly-primed and doubly-primed variables refer to the first and second symbol, respectively, on the right-hand side of the rule. Therefore, we rewrite the rule above as

$$\gamma_1 \hookrightarrow \gamma_2\gamma_3 \quad g' = 0 \wedge x' = x + 1 \wedge x'' = x \ .$$

We delay the formal definition of the relations to Section 4.3, where we describe how they can be encoded into bounded idempotent semirings, given bounded numbers of Boolean variables. Note that by using strings of Boolean variables we can represent any type of variables (e.g. integer or pointer) if it has a finite range. In Chapter 5 we compare two different representations of relations, bit vectors and BDDs, and discuss their advantages and disadvantages.

4.2.3 Java virtual machine modeling: Basics

Given an initial method where a reachability analysis should start, we first analyze which classes are statically reachable from the method. Then, bytecode instructions of these classes are translated into a pushdown system. The translation process is rather straightforward, since in most cases a bytecode instruction is mapped into a single weighted rule. Recall that stack symbols are constructed from method names and instruction offsets, and rules are always of the forms described in Section 4.2.1.

Therefore, given a bytecode instruction, we focus mainly in this section on translating the instruction into a weighted rule that models the behavior of the instruction as executed by the JVM. Although modeling control flow is a simple task, modeling variables is delicate, since one must fully understand the behavior of all bytecode instructions. We proceed step by step, ignoring some details at the beginning and only introducing them afterward.

Data types

Later on, we will need to differentiate between different data types. For the reference types, this is done by assigning a unique number—an *id*—to each class, interface, and array type that is reachable from the initial method. These numbers are then used to refer to their corresponding types. We also

Variables	Symbols	Types
Static variables	-	Globals
Heap	`h`	Array of globals
Heap pointer	`hp`	Global
Operand stack	`s`	Array of locals
Operand stack pointer	`sp`	Local
Local variable i	$\texttt{lv}_i$	Local

Table 4.2: Java variables including their notations and types

collect hierarchical information of types, which enables us to determine e.g. all superclasses of a class or all classes that implement an interface. Modeling of instructions such as `checkcast` and `invokevirtual` relies on this kind of information.

To ease the following presentation, we will ignore all primitive types, and always treat them as numbers. Section 4.2.4 discusses an extension.

Variables

The translator needs two inputs from users: the default number of *bits* that all variables should have and a *heap* length. With this information, we maintain four types of variables when translating bytecode instructions. Table 4.2 summarizes all variables and the symbols we use to identify them. Static variables and local variables (`lv`) are modeled by globals and locals, respectively. An operand stack (`s`) is modeled by an array of locals plus a *stack pointer* (`sp`), which is initialized to zero after method each invocation, and always points to the next available element in the array. The array length is equal to the maximum stack depth, which is predetermined by the compiler during compile time.

Similarly, the heap (`h`) is modeled by an array of globals (having the length given by the user) plus an extra *heap pointer* (`hp`). The heap pointer always points to the next available element in the array, and is initialized to one. We reserve the index zero for null objects. When an object is created, it occupies some parts of the array starting from where the heap pointer is pointing to. The object itself can be seen as a pointer to the array. The number of array elements an object occupies depends on its *size*. The size of an object, on the other hand, depends on object-specific information including the number of

Instance of a class with n instance fields:

class id	field$_1$	$\cdots$	field$_n$

Array a with length l:

array id	array length (l)	$a[0]$	$\cdots$	$a[l-1]$

Figure 4.8: Object information that is stored in the heap

instance fields it has. Each object has its own copy of instance fields, which we need to store in the array.

Figure 4.8 illustrates our design over the formats of objects in the heap. We differentiate two types of objects: class instance and array. Both begin with ids determining types. The next elements store instance fields in case of a class instance. For this, we need to assign a unique offset starting from zero to each field, and refer to them only by their offsets. For instance, if an instance `o` has 3 instance fields, namely `a`, `b`, and `c`, we might assign $0, 1, 2$ to `a`, `b`, `c`, respectively. To access `o.c`, we will need to access $\mathtt{h}[\mathtt{o} + 1 + 2]$. (Recall that objects are pointers, and we always need to add 1 for the ids.) For arrays, we need another element for array lengths before their actual array contents.

Translating bytecode instructions

We present in this section the translations for bytecode instructions. Recall from Section 4.1.2 that some bytecode instructions are grouped together because of their similar behaviors, e.g. the way the operand stack or the heap are operated on. We use this grouping again in the following, and only present a translation for each group. Other instructions in the same group can be translated in a similar manner.

We assume that all bytecode instructions are at the program point `p` and the next program point is `n`. Therefore, rules usually have the form

$$\mathtt{p} \hookrightarrow \mathtt{n} \quad R$$

and we only need to describe the relation R. Moreover, we assume that all other variables, which are not explicitly mentioned in the description of R

retain their values, i.e. if the variable $\mathtt{v}$ does not appear in an expression, we assume that $\mathtt{v}' = \mathtt{v}$.

Push Push constant x onto the operand stack.

$$\mathtt{s}'[\mathtt{sp}] = x \wedge \mathtt{sp}' = \mathtt{sp} + 1$$

Load Load local variable i onto the the operand stack.

$$\mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_i \wedge \mathtt{sp}' = \mathtt{sp} + 1$$

Store Store the value from the operand stack into local variable i.

$$\mathtt{lv}'_i = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1$$

Globalload Load static field f onto the operand stack.

$$\mathtt{s}'[\mathtt{sp}] = f \wedge \mathtt{sp}' = \mathtt{sp} + 1$$

Globalstore Store the value from the operand stack into static field f.

$$f' = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1$$

Unary Perform unary operation u with the value on the operand stack.

$$\mathtt{s}'[\mathtt{sp} - 1] = u(\mathtt{s}[\mathtt{sp} - 1])$$

Binary Perform binary operation $\star$ with the values on the operand stack.

$$\mathtt{s}'[\mathtt{sp} - 2] = \mathtt{s}[\mathtt{sp} - 2] \star \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1$$

Inc Increment local variable i by x.

$$\mathtt{lv}'_i = \mathtt{lv}_i + x$$

If Branch to label b if the comparison $\preceq$ between the value on the operand stack and constant c succeeds. We have to produce more than one rule in this case—one for each branch. Let $\succ$ be the complement operator of $\preceq$.

$$\begin{array}{ll} \mathtt{p} \hookrightarrow \mathtt{b} & \mathtt{s}[\mathtt{sp} - 1] \preceq c \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\ \mathtt{p} \hookrightarrow \mathtt{n} & \mathtt{s}[\mathtt{sp} - 1] \succ c \wedge \mathtt{sp}' = \mathtt{sp} - 1 \end{array}$$

Ifcmp Branch to label b if the comparison $\preceq$ between two values on the operand stack succeeds. We have to produce more than one rule in this case—one for each branch. Let $\succ$ be the complement operator of $\preceq$.

$$\begin{array}{ll} \mathtt{p} \hookrightarrow \mathtt{b} & \mathtt{s}[\mathtt{sp}-2] \preceq \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-2 \\ \mathtt{p} \hookrightarrow \mathtt{n} & \mathtt{s}[\mathtt{sp}-2] \succ \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-2 \end{array}$$

Goto Unconditionally branch to label `b`.

$$\mathtt{p} \hookrightarrow \mathtt{b}$$

As an example, the method in Figure 4.1 can be translated to the pushdown system having $\{\mathtt{local}_i \mid i \in \{0,1,2,5,8,9,11,14\}\}$ as the stack alphabet and containing the following weighted rules:

$$\begin{array}{lcll} \mathtt{local}_0 & \hookrightarrow & \mathtt{local}_1 & \mathtt{s}'[\mathtt{sp}] = 0 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\ \mathtt{local}_1 & \hookrightarrow & \mathtt{local}_2 & \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-1 \\ \mathtt{local}_2 & \hookrightarrow & \mathtt{local}_8 & \\ \mathtt{local}_5 & \hookrightarrow & \mathtt{local}_8 & \mathtt{lv}'_0 = \mathtt{lv}_0 + 1 \\ \mathtt{local}_8 & \hookrightarrow & \mathtt{local}_9 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\ \mathtt{local}_9 & \hookrightarrow & \mathtt{local}_{11} & \mathtt{s}'[\mathtt{sp}] = 10 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\ \mathtt{local}_{11} & \hookrightarrow & \mathtt{local}_5 & \mathtt{s}[\mathtt{sp}-2] < \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-2 \\ \mathtt{local}_{11} & \hookrightarrow & \mathtt{local}_{14} & \mathtt{s}[\mathtt{sp}-2] \geq \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-2 \\ \mathtt{local}_{14} & \hookrightarrow & \varepsilon & \end{array}$$

Invoke Invoke a method with `m` as its entry point. Obviously, this corresponds to a push rule.

$$\mathtt{p} \hookrightarrow \mathtt{m}\ \mathtt{n} \quad \mathtt{sp}' = 0$$

The expression $\mathtt{sp}' = 0$ initializes the stack pointer of `m` to zero. Recall that method arguments are passed from the operand stack to local variables of the new method. Therefore, if the method has $n > 0$ arguments, we need to append the following expression:

$$\mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp}-n] \wedge \ldots \wedge \mathtt{lv}'_{n-1} = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = 0 \wedge \mathtt{sp}'' = \mathtt{sp}-n\ .$$

Note that in case of instance or interface method invocations, given a class and method name, the method to be invoked depends on the instance type of the reference (the n-th element on the operand stack). For this, we need to find all candidate methods that can be invoked, each method corresponds to a *candidate set* of class ids. The candidate methods can be found by searching for the invoked method up and down in the class hierarchy. For instance, let the method to be invoked be `size()I` of class `java.util.AbstractList` (having, say, id 1), and assume that there are three other classes of interest: `java.util.Vector` (id 2), `java.util.Stack` (id 3), and `java.util.ArrayList` (id 4). The classes `Vector` and `ArrayList` are subclasses of `AbstractList`, while `Stack` is in turn a subclass of `Vector`. Since only `Vector` and `ArrayList` define the method `size()I` in this case, we have two candidate methods, one for each class, with the corresponding sets $\{2, 3\}$ and $\{4\}$, respectively.

Then, we create a rule for each candidate method that checks for the membership of the instance type in the corresponding candidate sets in order to invoke the right method. Assume that there are j candidate methods with entry points $m_1, \ldots, m_j$ corresponding to candidate sets $S_1, \ldots, S_j$. We introduce j rules of the following form, for $1 \leq i \leq j$ (omitting argument passing):

$$\mathtt{p} \hookrightarrow \mathtt{m}_i\ \mathtt{n} \quad \mathtt{h}[\mathtt{s}[\mathtt{sp} - n]] \in S_i \wedge \ldots$$

Continuing the example above, we construct the following two rules, one for each candidate. Assume that $\mathtt{Vector.size}_0$ and $\mathtt{ArrayList.size}_0$ are the entry points of the methods `size()I` in `Vector` and `ArrayList`, respectively:

$$\begin{array}{lcll} \mathtt{p} & \hookrightarrow & \mathtt{Vector.size}_0\ \mathtt{n} & \mathtt{h}[\mathtt{s}[\mathtt{sp} - 1]] \in \{2, 3\} \wedge \ldots \\ \mathtt{p} & \hookrightarrow & \mathtt{ArrayList.size}_0\ \mathtt{n} & \mathtt{h}[\mathtt{s}[\mathtt{sp} - 1]] \in \{4\} \wedge \ldots \end{array}$$

If the invoked method returns a value, we introduce a fresh stack symbol `f` as a new return address for storing the return value from `ret` onto the operand stack before continuing (cf. Return).

$$\begin{array}{lcll} \mathtt{p} & \hookrightarrow & \mathtt{m}_i\ \mathtt{f} & \ldots \\ \mathtt{f} & \hookrightarrow & \mathtt{n} & \mathtt{s}'[\mathtt{sp}] = \mathtt{ret} \wedge \mathtt{sp}' = \mathtt{sp} + 1 \end{array}$$

Return Method return corresponds to a pop rule.

$$\mathtt{p} \hookrightarrow \varepsilon$$

If the method returns a value, we store it in a temporary global variable `ret`. The value of `ret` will be retrieved later on by the invoker method (cf.

Invoke).

$$\mathtt{ret}' = \mathtt{s}[\mathtt{sp}-1]$$

The example in Figure 4.2 can be translated to a pushdown system containing the following weighted rules (Notice the use of the fresh stack symbol $\mathtt{tt_f}$ for restoring the return value.):

$$\begin{array}{lcll}
\mathtt{sub}_0 & \hookrightarrow & \mathtt{sub}_1 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\
\mathtt{sub}_1 & \hookrightarrow & \mathtt{sub}_2 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_1 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\
\mathtt{sub}_2 & \hookrightarrow & \mathtt{sub}_3 & \mathtt{s}'[\mathtt{sp}-2] = \mathtt{s}[\mathtt{sp}-2] - \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-1 \\
\mathtt{sub}_3 & \hookrightarrow & \varepsilon & \mathtt{ret}' = \mathtt{s}[\mathtt{sp}-1] \\
\mathtt{tt}_0 & \hookrightarrow & \mathtt{tt}_1 & \mathtt{s}'[\mathtt{sp}] = 3 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\
\mathtt{tt}_1 & \hookrightarrow & \mathtt{tt}_2 & \mathtt{s}'[\mathtt{sp}] = 2 \wedge \mathtt{sp}' = \mathtt{sp}+1 \\
\mathtt{tt}_2 & \hookrightarrow & \mathtt{sub}_0\ \mathtt{tt_f} & \mathtt{sp}' = 0 \wedge \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp}-2] \wedge \mathtt{lv}'_1 = \mathtt{s}[\mathtt{sp}-1] \\
 & & & \quad \wedge\ \mathtt{sp}'' = \mathtt{sp}-2 \\
\mathtt{tt_f} & \hookrightarrow & \mathtt{tt}_5 & \mathtt{s}'[\mathtt{sp}] = \mathtt{ret} \wedge \mathtt{sp}' = \mathtt{sp}+1 \\
\mathtt{tt}_5 & \hookrightarrow & \varepsilon & \mathtt{ret}' = \mathtt{s}[\mathtt{sp}-1]
\end{array}$$

Newarray Create a one-dimensional array of type *id* with length determined by the value on top of the operand stack.

$$\mathtt{h}'[\mathtt{hp}] = id \wedge \mathtt{h}'[\mathtt{hp}+1] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{s}'[\mathtt{sp}-1] = \mathtt{hp} \wedge \mathtt{hp}' = \mathtt{hp}+2+\mathtt{s}[\mathtt{sp}-1]$$

Note that a multi-dimensional array can be created similarly by using multiple copies of one-dimensional arrays.

Arraylength Load the array length of the array specified by the top element of the operand stack onto the operand stack.

$$\mathtt{s}'[\mathtt{sp}-1] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+1]$$

Arrayload Load the array element at index specified by the top-of-stack value onto the operand stack. The second element on the operand stack specifies the array.

$$\mathtt{s}'[\mathtt{sp}-2] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-2]+2+\mathtt{s}[\mathtt{sp}-1]] \wedge \mathtt{sp}' = \mathtt{sp}-1$$

Arraystore Store the value on top of the operand stack into an array at index specified by the second element on the operand stack. The third element on the operand stack specifies the array.

$$\mathtt{h}'[\mathtt{s}[\mathtt{sp}-3]+2+\mathtt{s}[\mathtt{sp}-2]] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-3$$

The method ar in Figure 4.3 can be translated to a pushdown system containing the following weighted rules. Assume that the array has id 1.

$\mathtt{ar}_0$	$\hookrightarrow$	$\mathtt{ar}_1$	$\mathtt{s}'[\mathtt{sp}] = 2 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_1$	$\hookrightarrow$	$\mathtt{ar}_3$	$\mathtt{h}'[\mathtt{hp}] = 1 \wedge \mathtt{h}'[\mathtt{hp}+1] = \mathtt{s}[\mathtt{sp}-1]$ $\wedge\ \mathtt{s}'[\mathtt{sp}-1] = \mathtt{hp} \wedge \mathtt{hp}' = \mathtt{hp}+2+\mathtt{s}[\mathtt{sp}-1]$
$\mathtt{ar}_3$	$\hookrightarrow$	$\mathtt{ar}_4$	$\mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-1$
$\mathtt{ar}_4$	$\hookrightarrow$	$\mathtt{ar}_5$	$\mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_5$	$\hookrightarrow$	$\mathtt{ar}_6$	$\mathtt{s}'[\mathtt{sp}] = 0 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_6$	$\hookrightarrow$	$\mathtt{ar}_7$	$\mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_7$	$\hookrightarrow$	$\mathtt{ar}_8$	$\mathtt{s}'[\mathtt{sp}-1] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+1]$
$\mathtt{ar}_8$	$\hookrightarrow$	$\mathtt{ar}_9$	$\mathtt{h}'[\mathtt{s}[\mathtt{sp}-3]+2+\mathtt{s}[\mathtt{sp}-2]] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-3$
$\mathtt{ar}_9$	$\hookrightarrow$	$\mathtt{ar}_{10}$	$\mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_{10}$	$\hookrightarrow$	$\mathtt{ar}_{11}$	$\mathtt{s}'[\mathtt{sp}] = 1 \wedge \mathtt{sp}' = \mathtt{sp}+1$
$\mathtt{ar}_{11}$	$\hookrightarrow$	$\mathtt{ar}_{12}$	$\mathtt{s}'[\mathtt{sp}-2] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-2]+2+\mathtt{s}[\mathtt{sp}-1]] \wedge \mathtt{sp}' = \mathtt{sp}-1$
$\mathtt{ar}_{12}$	$\hookrightarrow$	ε	$\mathtt{ret}' = \mathtt{s}[\mathtt{sp}-1]$

New Create a new class instance of type *id* having *n* instance fields.

$$\mathtt{h}'[\mathtt{hp}] = id \wedge \mathtt{s}'[\mathtt{sp}] = \mathtt{hp} \wedge \mathtt{hp}' = \mathtt{hp}+1+n \wedge \mathtt{sp}' = \mathtt{sp}+1$$

Fieldload Load the field at offset *f* of the class instance specified by the top element of the operand stack.

$$\mathtt{s}'[\mathtt{sp}-1] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+1+f]$$

Fieldstore Store the value on top of the operand stack to the field at offset *f* of the class instance specified by the second top element of the operand stack.

$$\mathtt{h}'[\mathtt{s}[\mathtt{sp}-2]+1+f] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp}-2$$

Instanceof Check whether the object specified by the reference on top of the operand stack is of a given type. Similar to instance method invocations, we need to precompute a candidate set S, and check whether the object's id belongs to S. The candidate set contains ids of all types that the object can possibly have, which can be statically computed given types of interest. For instance, if the type is an ordinary (nonarray) class, the candidate set will contain its id and the ids of all subclasses. Therefore, a check whether an object is of a given type boils down to checking whether the object's id is a member of the candidate set. See [41] for a complete reference of how to constructs the candidate sets for different types. The bytecode instruction `instanceof` pushes an integer 1, if the check succeeds or 0, otherwise.

$$(\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \in S \wedge \mathtt{s}'[\mathtt{sp}-1] = 1) \vee (\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \notin S \wedge \mathtt{s}'[\mathtt{sp}-1] = 0)$$

Pop Pop the operand stack x times.

$$\mathtt{sp}' = \mathtt{sp} - x$$

Dup Duplicate elements on the operand stack. There are six different duplication instructions, and some of their behaviors depend on types of values on the operand stack. We will not elaborate on this, but only point out that to model all behaviors correctly, we need information on variable types discussed in Section 4.2.4. Nevertheless, the bytecode instruction `dup`, which always duplicates a single element on the operand stack, is perhaps the most often used.

$$\mathtt{s}'[\mathtt{sp}] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp} + 1$$

We consider again the example in Figure 4.4. The class `B` has one instance field (with offset 0), namely `x`. Hence, given a reference r to an instance of class `B` the value of `B.x` can be accessed by the expression $\mathtt{h}[r+1+0]$. Assume that the ids of the classes `A` and `B` are 1 and 2, respectively. Since `A` is an ordinary class, its candidate set contains the id of `A` itself and the ids of all subclasses of `A`, i.e. $\{1, 2\}$ in this case. Let $\mathtt{init}_0$ be the entry point of the constructor `B.<init>()V`. The method `in` can be translated to a weighted pushdown system containing the following weighted rules:

$$\begin{array}{lll}
\mathtt{in}_0 & \hookrightarrow & \mathtt{in}_3 & \mathtt{h}'[\mathtt{hp}] = 2 \wedge \mathtt{s}'[\mathtt{sp}] = \mathtt{hp} \\
 & & & \quad \wedge\, \mathtt{hp}' = \mathtt{hp} + 1 + 1 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{in}_3 & \hookrightarrow & \mathtt{in}_4 & \mathtt{s}'[\mathtt{sp}] = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{in}_4 & \hookrightarrow & \mathtt{init}_0\ \mathtt{in}_7 & \mathtt{sp}' = 0 \wedge \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}'' = \mathtt{sp} - 1 \\
\mathtt{in}_7 & \hookrightarrow & \mathtt{in}_8 & \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{in}_8 & \hookrightarrow & \mathtt{in}_9 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{in}_9 & \hookrightarrow & \mathtt{in}_{12} & (\mathtt{h}[\mathtt{s}[\mathtt{sp} - 1]] \in \{1, 2\} \wedge \mathtt{s}'[\mathtt{sp} - 1] = 1) \\
 & & & \quad \vee\, (\mathtt{h}[\mathtt{s}[\mathtt{sp} - 1]] \notin \{1, 2\} \wedge \mathtt{s}'[\mathtt{sp} - 1] = 0) \\
\mathtt{in}_{12} & \hookrightarrow & \mathtt{in}_{20} & \mathtt{s}[\mathtt{sp} - 1] = 0 \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{in}_{12} & \hookrightarrow & \mathtt{in}_{15} & \mathtt{s}[\mathtt{sp} - 1] \neq 0 \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{in}_{15} & \hookrightarrow & \mathtt{in}_{16} & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{in}_{16} & \hookrightarrow & \mathtt{in}_{17} & \mathtt{s}'[\mathtt{sp}] = 1 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{in}_{17} & \hookrightarrow & \mathtt{in}_{20} & \mathtt{h}'[\mathtt{s}[\mathtt{sp} - 2] + 1 + 0] = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 2 \\
\mathtt{in}_{20} & \hookrightarrow & \varepsilon &
\end{array}$$

Swap Swap the top two operand stack values.

$$\mathtt{s}'[\mathtt{sp} - 1] = \mathtt{s}[\mathtt{sp} - 2] \wedge \mathtt{s}'[\mathtt{sp} - 2] = \mathtt{s}[\mathtt{sp} - 1]$$

Wrapper

Before analyzing a method, in addition to translating the methods that are statically reachable as described above, we need to create a special method, called *wrapper*. Basically, the wrapper initializes variables, in particular the heap, and invoke the initial method. Accordingly, reachability analyses always start from the wrapper. The following expression initializes the heap length l:

$$\mathtt{hp}' = 1 \wedge \bigwedge_{i=0}^{l-1} \mathtt{h}'[i] = 0 \ .$$

Moreover, sometimes one might want to test whether a method always works correctly within a given input range, e.g. whether a sorting implementation always correctly returns sorted arrays or whether an exception can be thrown. We also use wrappers for this purpose. Given a initial method the wrapper wraps the method by calling it with nondeterministic argument values.

For example, to test the method `m(int a, int[] b)`, where all values can be either 0 or 1, and `b` has length at most 3, we can wrap it with the following rules:

$$
\begin{array}{lcll}
\mathtt{w}_0 & \hookrightarrow & \mathtt{w}_1 & \mathtt{sp}' = 0 \wedge \mathtt{hp}' = 1 \wedge \bigwedge_{i=0}^{l-1} \mathtt{h}'[i] = 0 \\
\mathtt{w}_1 & \hookrightarrow & \mathtt{w}_2 & 0 \leq \mathtt{s}'[\mathtt{sp}] \leq 1 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{w}_2 & \hookrightarrow & \mathtt{w}_3 & 0 \leq \mathtt{s}'[\mathtt{sp}] \leq 3 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{w}_3 & \hookrightarrow & \mathtt{w}_4 & \mathtt{h}'[\mathtt{hp}] = id \wedge \mathtt{h}'[\mathtt{hp}+1] = \mathtt{s}[\mathtt{sp}-1] \\
 & & & \quad \wedge\, \mathtt{s}'[\mathtt{sp}-1] = \mathtt{hp} \wedge \mathtt{hp}' = \mathtt{hp} + 2 + \mathtt{s}[\mathtt{sp}-1] \\
 & & & \quad \wedge \bigwedge_{i=0}^{\mathtt{s}[\mathtt{sp}-1]-1} 0 \leq \mathtt{h}'[\mathtt{hp}+2+i] \leq 1 \\
\mathtt{w}_4 & \hookrightarrow & \mathtt{m}_0\ \mathtt{w}_5 & \mathtt{sp}' = 0 \wedge \bigwedge_{i=0}^{1} \mathtt{lv}'_i = \mathtt{s}[\mathtt{sp}-2+i] \wedge \mathtt{sp}'' = \mathtt{sp} - 2\ ,
\end{array}
$$

where *id* specifies the array type. The analysis should then start from $\mathtt{w}_0$.

4.2.4 Java virtual machine modeling: Extensions

In this section we present some extensions needed for supporting more Java features omitted from the model in the last section. We will not list all needed changes, but will only explain and justify the ideas of possible extensions. All extensions have been implemented in our tool, presented in Chapter 5.

Category 2 computational type

Although the Java virtual machine specification specifies types of variables, on which many instructions must strictly operate, we have chosen to ignore them so far. For instance, consider the bytecode instructions `astore_0` and `istore_0`, which pop an object reference and an integer, respectively, from the operand stack and store it into the local variable 0. In the previous section (cf. Store), we treat these two instructions exactly the same by popping *whatever* is on the operand stack and storing it into the local variable 0.

The resulting models behave correctly in the case where variable types are well-defined by bytecode instructions (most of them are), e.g. in `getfield` (cf. Fieldload) the value on the operand stack is always a reference to a class instance. For these instructions, we know which variable types are expected in which positions on the operand stack. Obviously, here we assume that the instructions are always correctly typed, so that illegal type conversions, such as converting integers to object references, are not possible. Class files obtained directly from Sun's compiler are known to satisfy this constraint [41].

Nevertheless, [41] specifies long and double—so called *category 2 computational type*—differently. Most notably, each long or double occupies *two* consecutive local variables. For instance, `lload_1` pushes *a* long value specified by the local variables 1 and 2 onto the operand stack. Moreover, some bytecode instructions such as `dup2` behave differently depending on categories of values on the operand stack. This poses a problem in the design described in the last section, where a value always occupies only one local variable.

Our model, however, can be easily extended to support these types. We need to differentiate between the instructions that operate on category 1 computational types (all other types except long and double) and category 2 computational types; thus introducing more expression types that must be taken care of. Also, we use two stack elements for a value of category 2. This decision choice is natural, since we can maintain one-to-one relationship between stack elements and local variables. The operations such as argument passing are easy to handle, because stack elements are simply copied one by one to corresponding local variables of the invoked method.

Exceptions

When an exception is thrown, we need to search for its exception handler as described in Section 4.1.1. Similar to modeling instance method invocations described in the previous section, this behavior is modeled by enumerating rules for all possible exception handlers. For each exception handler, we construct a candidate set containing exception types that can be handled. Assume that the stack symbol `p` corresponds to the location of `athrow`—the bytecode instruction that throws the exception specified by the element on the operand stack—and there are j possible exception handlers that handle exceptions $S_1, \ldots, S_j$ by branching to locations $\mathtt{b}_1, \ldots, \mathtt{b}_j$, respectively. For each $1 \leq i \leq j$, we model `athrow` by the following rules. Notice that `athrow` clears the operand stack before pushing the exception instance back.

$$\mathtt{p} \hookrightarrow \mathtt{b}_i \quad \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \in S_i \wedge \mathtt{s}'[0] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = 1$$

Moreover, we need to propagate all exceptions when their handlers are not present in the current method. Exception propagations can be modeled by an extra global variable `e`, indicating the current exception state. Its value can be either zero, meaning that no exception has been thrown, or a (non-zero) *id*, signifying the class that has been thrown. The bytecode instruction `athrow` updates the value of `e` with the value on the operand stack when no exception handlers are found in the current method.

$$\mathtt{p} \hookrightarrow \varepsilon \quad \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \notin \bigcup_{i=1}^{j} S_i \wedge \mathtt{e}' = \mathtt{s}[\mathtt{sp}-1]$$

Obviously, we need to check the value of `e` after each instruction, including in particular method invocations, that can throw exceptions. Assume that `p` is such a location. Again, we enumerate all possible exceptions $S_1, \ldots, S_j$ at that location, and branch to their corresponding handlers $\mathtt{b}_1, \ldots, \mathtt{b}_j$ defined in the current method. For $1 \leq i \leq j$, we have

$$\mathtt{p} \hookrightarrow \mathtt{b}_i \quad \mathtt{e} \in S_i \wedge \mathtt{e}' = 0 \wedge \mathtt{s}'[0] = \mathtt{e} \wedge \mathtt{sp}' = 1$$

In the case where the handler is not found in the current method, the exception must propagate further.

$$\mathtt{p} \hookrightarrow \varepsilon \quad \mathtt{e} \neq 0 \wedge \mathtt{e} \notin \bigcup_{i=1}^{j} S_i$$

Consider again the example in Figure 4.5. Let $\mathtt{init}_0$ be the entry point of the constructor `java/lang/RuntimeException.<init>()V`. By assuming that the class `Exception` has id 1, `RuntimeException` has id 2, and both containing no instance fields, the method `ex` can be translated to a weighted pushdown system containing the following weighted rules:

$\mathtt{ex}_0$	$\hookrightarrow$	$\mathtt{ex}_3$	$\mathtt{h}'[\mathtt{hp}] = 2 \wedge \mathtt{s}'[\mathtt{sp}] = \mathtt{hp}$ $\wedge\, \mathtt{hp}' = \mathtt{hp} + 1 + 0 \wedge \mathtt{sp}' = \mathtt{sp} + 1$
$\mathtt{ex}_3$	$\hookrightarrow$	$\mathtt{ex}_4$	$\mathtt{s}'[\mathtt{sp}] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp} + 1$
$\mathtt{ex}_4$	$\hookrightarrow$	$\mathtt{init}_0\ \mathtt{ex}_7$	$\mathtt{sp}' = 0 \wedge \mathtt{lv}_0' = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}'' = \mathtt{sp} - 1$
$\mathtt{ex}_7$	$\hookrightarrow$	$\mathtt{ex}_8$	$\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \in \{1,2\} \wedge \mathtt{s}'[0] = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = 1$
$\mathtt{ex}_7$	$\hookrightarrow$	ε	$\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]] \notin \{1,2\} \wedge \mathtt{e}' = \mathtt{s}[\mathtt{sp}-1]$
$\mathtt{ex}_8$	$\hookrightarrow$	$\mathtt{ex}_9$	$\mathtt{lv}_0' = \mathtt{s}[\mathtt{sp}-1] \wedge \mathtt{sp}' = \mathtt{sp} - 1$
$\mathtt{ex}_9$	$\hookrightarrow$	$\mathtt{ex}_{10}$	$\mathtt{s}[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1$
$\mathtt{ex}_{10}$	$\hookrightarrow$	ε	$\mathtt{e}' = \mathtt{s}[\mathtt{sp}-1]$

finally

As specified in [41], finally blocks are compiled into subroutines inside methods. Two special bytecode instructions are used: `jsr` (jump to subroutine) and `ret` (return from subroutine). The `jsr` instruction invokes a subroutine, and pushes the return address—the address of the instruction immediately following `jsr`—onto the operand stack. Then, the return address is stored inside the subroutine into a local variable. At the end of subroutine, `ret` fetches the return address from the local variable and transfers the control to the instruction at the return address. Obviously, this behavior can be modeled by mapping all labels after `jsr` to unique numbers. These numbers are used to push into the operand stack when modeling `jsr`. When translating `ret`, we just need to enumerate all these numbers in order to branch to the right return address.

However, these two bytecode instructions are no longer supported by Java 6.0 as they will be rejected by its type checking verifier [43]. Somewhat informally, Sun's compiler also no longer generate them. Instead, it inlines the finally code to all possible exit points to guarantee that the code will always be executed.

For this reason, we will not investigate this issue in any further detail.

Multithreading

If a program involves more than one thread, we instead translate it to a pushdown network, to which the context-bounded reachability analysis in Section 3.1.3 can be applied. The set of shared globals obviously contains the heap and all class instances. Recall, however, that the reachability algorithm is always initialized with a global configuration consisting of a fixed number of threads. In Java, programs are started with one thread, and threads can be dynamically created. For this, we extend the control flow modeling in Section 4.2.1 with so-called *dynamic rules*

$$\gamma \hookrightarrow \gamma' \rhd \gamma'' ,$$

which transforms the program point γ to γ', and forks a new thread starting with the program point γ''. We defer the formal definition of dynamic rules and the extension of the reachability algorithm to Section 4.3.2. For now, we are only interested in translating bytecode instructions to pushdown networks with dynamic thread creations.

The translation discussed in the previous section is still applicable in most cases, except that we need to store more information in the heap for modeling synchronization. Recall from Section 4.1.1 that each object has a monitor associated with it. Therefore, the object information in Figure 4.8

Instance of a class with n instance fields:

class id	monitor owner	monitor counter	field$_1$	$\cdots$	field$_n$

Array a with length l:

array id	monitor owner	monitor counter	array length (l)	$a[0]$	$\cdots$	$a[l-1]$

Figure 4.9: Object information for multithreaded programs

needs to be extended to incorporate two pieces of information concerning monitors—*monitor owner* and *monitor counter*. Figure 4.9 shows our design. We assign to each thread a unique id starting from one. The monitor owner of an object stores the thread id that currently owns the monitor, and the number of times the thread acquired the monitor is stored in the monitor counter. Translations of bytecode instructions listed in the previous section involving objects must be adjusted to include the fact that sizes of objects are now increased by two. There are two bytecode instructions that work on monitors.

Monitorenter Enter or reenter the monitor of the object specified by the reference on the operand stack. We assume that the variable `tid` stores the current thread id. Monitorenter succeeds only if no other thread is currently holding the monitor.

$$\begin{array}{rl} & (\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+1] = \mathtt{tid} \vee \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+2] = 0) \\ \wedge & \mathtt{h}'[\mathtt{s}[\mathtt{sp}-1]+1] = \mathtt{tid} \\ \wedge & \mathtt{h}'[\mathtt{s}[\mathtt{sp}-1]+2] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+2]+1 \end{array}$$

Monitorexit Exit the monitor of the object specified by the reference on the operand stack.

$$\begin{array}{rl} & \mathtt{h}'[\mathtt{s}[\mathtt{sp}-1]+2] = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+2]-1 \\ \wedge & (\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+2] = 1 \Rightarrow \mathtt{h}'[\mathtt{s}[\mathtt{sp}-1]+1] = 0) \end{array}$$

Recall from Section 4.1.1 that there are two approaches to fork a new thread: either directly create a subclass of `Thread` or implements the interface

Runnable and pass it as an argument when constructing an object of type Thread. Then, a new thread can be forked by calling the method start. Both approaches can be handled similarly, so we discuss only the second approach in the following. Recall also that the bytecode instruction invokevirtual might start a new thread if the invoked method is the specialized method used by Java for starting new threads, i.e. the private native method start0() of the class Thread. We model thread creations similarly to modeling virtual method invocations by enumerating dynamic rules for all possible candidate classes that implement the interface Runnable. As specified by Java, for each enumeration i the starting point $\mathtt{m}_i$ of the new thread is the run() method of the corresponding candidate class. Moreover, we employ the fact that the class instance that implements Runnable is a field of Thread. Assume that the field has offset 0, and therefore we can write $\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+3+0]$ to access it (cf. Fieldload). The class instance is passed as the only argument. With the convention that the doubly-primed variables refer to the variables of the new thread, we have

$$\begin{array}{lll} \mathtt{p} \hookrightarrow \mathtt{n} \rhd \mathtt{m}_i & & \mathtt{h}[\mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+3+0]] = id_i \\ & & \wedge\ \mathtt{lv}''_0 = \mathtt{h}[\mathtt{s}[\mathtt{sp}-1]+3+0] \wedge \mathtt{sp}'' = 0 \\ & & \wedge\ \mathtt{sp}' = \mathtt{sp}-1 \end{array}$$

Figure 4.10 gives an example of a pushdown network translated from the bytecode instructions in Figure 4.6. The first four rules allocate two objects of types Thread and C$1. We assume here that Thread and C$1 have type 1 and 2, respectively. The Thread object is simplified from the original Java implementation to contain only one field of type Runnable, which is the target object where the run method is implemented. The C$1 object is initialized at offset 8 by a push rule with $\mathtt{c}_0$ as the entry point and a reference to the object as the only argument. Similarly, at offset 11 the Thread object is initialized with $\mathtt{t}_0$ as the entry point, but this time both object references are passed as the arguments. The reference to C$1 is stored as the the only field of the Thread object (rules not shown).

The method start (abbreviated with s) is called at offset 14 with $\mathtt{s}_0$ as the entry point. It later forks a new thread at offset 29, which corresponds to a dynamic rule. Note that, in this case, there is only one class that implements Runnable, namely C$1, so there is only one dynamic rule. Obviously, the check whether the instance has type 2 is always true and $\mathtt{r}_0$ is always the entry point of the new thread.

$$
\begin{array}{llll}
\mathtt{f}_0 & \hookrightarrow & \mathtt{f}_3 & \mathtt{h}'[\mathtt{hp}] = 1 \wedge \mathtt{s}'[\mathtt{sp}] = \mathtt{hp} \\
 & & & \quad \wedge\, \mathtt{hp}' = \mathtt{hp} + 3 + 1 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{f}_3 & \hookrightarrow & \mathtt{f}_4 & \mathtt{s}'[\mathtt{sp}] = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{f}_4 & \hookrightarrow & \mathtt{f}_7 & \mathtt{h}'[\mathtt{hp}] = 2 \wedge \mathtt{s}'[\mathtt{sp}] = \mathtt{hp} \\
 & & & \quad \wedge\, \mathtt{hp}' = \mathtt{hp} + 3 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{f}_7 & \hookrightarrow & \mathtt{f}_8 & \mathtt{s}'[\mathtt{sp}] = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{f}_8 & \hookrightarrow & \mathtt{c}_0\ \mathtt{f}_{11} & \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}'' = \mathtt{sp} - 1 \\
\mathtt{f}_{11} & \hookrightarrow & \mathtt{t}_0\ \mathtt{f}_{14} & \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp} - 2] \wedge \mathtt{lv}'_1 = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}'' = \mathtt{sp} - 2 \\
\mathtt{f}_{14} & \hookrightarrow & \mathtt{s}_0\ \mathtt{f}_{17} & \mathtt{lv}'_0 = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
 & \ldots & & \ldots \\
\mathtt{f}_e & \hookrightarrow & \varepsilon & \\
\mathtt{s}_0 & \hookrightarrow & \ldots & \ldots \\
\mathtt{s}_{28} & \hookrightarrow & \mathtt{s}_{29} & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{s}_{29} & \hookrightarrow & \mathtt{s}_{32} \rhd \mathtt{r}_0 & \mathtt{h}[\mathtt{h}[\mathtt{s}[\mathtt{sp} - 1] + 3 + 0]] = 2 \wedge \mathtt{sp}'' = 0 \\
 & & & \quad \wedge\, \mathtt{lv}''_0 = \mathtt{h}[\mathtt{s}[\mathtt{sp} - 1] + 3 + 0] \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
 & \ldots & & \ldots
\end{array}
$$

Figure 4.10: A pushdown network translated from the code in Figure 4.6

4.3 Applying the reachability analyses

We discuss in this section several issues that arise when applying the reachability analyses in Chapter 3 to the translated pushdown models generated by the translator described in the previous section.

4.3.1 Representing variable relations as semirings

In the previous section, we discussed how Java bytecode instructions are translated into pushdown rules where weights are relations between variables. We now focus on how these variable relations are represented in a bounded idempotent semiring, so that the algorithms in Sections 3.1.1 and 3.1.3—the reachability algorithms when weights are bounded idempotent semirings—are applicable.

When describing an expression, several copies of variables are usually needed, e.g. for $\texttt{sp}' = \texttt{sp} + 1$, two copies of `sp` are needed in order to represent the relation of values of `sp` *before* and *after* the expression. In the following, we will restrict ourselves only to bounded variables. Notice that by using strings of Boolean variables it is possible to represent bounded variables of any types (e.g. integer or pointer).

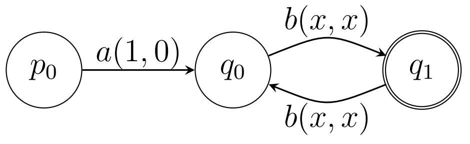
Assume that bounded globals and locals can be represented by m and n Boolean variables, respectively. We define globals and locals as

$$G = \{0,1\}^m \text{ and } L = \{0,1\}^n \ .$$

In the following, we present the missing connection between the bounded idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ and variable relations. Recall from Section 4.2.2 that weights for different types of rules contain different numbers of locals, depending on numbers of stack symbols on right-hand sides. Therefore, to simplify the presentation we proceed by considering first the case without locals.

Semiring without locals

When only global variables are considered, the weight domain can be defined as $D = 2^{G \times G}$. The combine operation is simply union with the empty set as neutral element. That is, given $R, S \in D$, computing $T = R \oplus S$ means computing $T = R \cup S$. On the other hand, we need to *join* two elements



Rules:

$\langle p_0, a\rangle$	$\hookrightarrow$	$\langle p_0, \varepsilon\rangle$	$R(x,x)$
$\langle p_0, a\rangle$	$\hookrightarrow$	$\langle p_1, a\rangle$	$R(x,1)$
$\langle p_1, a\rangle$	$\hookrightarrow$	$\langle p_0, ab\rangle$	$R(1,1)$

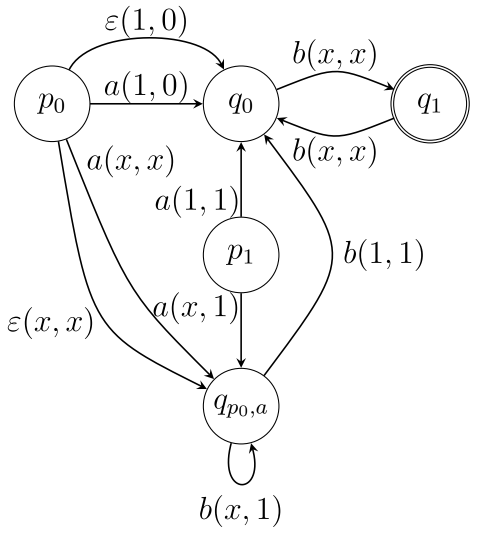


Figure 4.11: The weighted automata $\mathcal{WA}$ (left) and $\mathcal{WA}_{post^*}$ (right)

$R, S \in D$ when computing the extend operation, i.e. $T = R \otimes S$ is defined as follows:

$$T(g, g'') = \exists g' \in G : R(g, g') \wedge S(g', g'') \ .$$

The neutral element of $\otimes$ is the identity relation $I_G = \{(g, g) \mid g \in G\}$.

Intuitively, when a weight is used to annotate a rule, it means a relation (i.e. pairs) of variable valuations before and after the rule is executed. This idea has been expressed earlier in Section 4.2.2. Similarly, when annotating automaton transitions, these pairs can be considered as valuations of global variables of the connected states. Note, however, that the pairs are in reversed order of the transitions, i.e. given a transition (q, a, q') and a pair (g, g'), the valuation g belongs to the state q' and vice versa.

Figure 4.11—repeated from Figure 3.1, but with a different semiring—shows an example of weighted automata when only one global (Boolean) variable is present, i.e. $G = \{0, 1\}$. Every transition is labeled by a symbol followed by a weight. We write $(1, 0)$ instead of $\{(1, 0)\}$, and use $x \in \{0, 1\}$ as a shorthand, e.g. (x, x) means $\{(0, 0), (1, 1)\}$ and $(x, 1)$ means $\{(0, 1), (1, 1)\}$. These abbreviations are used similarly for weights of rules.

As an example, consider the configuration $\langle p_0, abbb\rangle$ accepted by both automata. On the left automaton $\mathcal{WA}$, $\langle p_0, abbb\rangle$ is accepted by following the path: $p_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_0 \xrightarrow{b} q_1$. Thus, its weight is

$$(x, x) \otimes (x, x) \otimes (x, x) \otimes (1, 0) = (1, 0) \ ,$$

which means that the global variable has value 0 in $\langle p_0, abbb\rangle$. The right automaton $\mathcal{WA}_{post^*}$, however, accepts $\langle p_0, abbb\rangle$ not only through the path above, but also through: $p_0 \xrightarrow{a} q_{p_0,a} \xrightarrow{b} q_{p_0,a} \xrightarrow{b} q_0 \xrightarrow{b} q_1$. This path contributes the weight

$$(x, x) \otimes (1, 1) \otimes (x, 1) \otimes (x, x) = (1, 1)$$

to the configuration. Thus, the weight of $\langle p_0, abbb\rangle$ on the right automaton is

$$(1, 0) \oplus (1, 1) = (1, x) \ ,$$

i.e. the global variable can be either 0 or 1.

Semiring with locals

Handling locals is not as easy as handling globals due to the fact that relations can be of different arities. Table 4.1 gives the first glimpse of such relations, where the arities depend on numbers of stack symbols on right-hand sides. That is, while a copy of locals is always needed on the left-hand side, zero, one, two copies are varyingly required for pop, normal, push and dynamic rules, respectively. Moreover, given a configuration $\langle p, w\rangle$, its weight should contain a copy of global valuations and $|w|$ copies of local valuations, one for each element on the configuration stack. (In our context, these copies represent the *current* global and local valuations together with local valuations of all return addresses.)

In short, the number of stack symbols determines the arity of a relation. It is therefore reasonable to include stack symbols into weights. We call these stack symbols *signatures*, or types, of the weights. Formally, the weight domain D consists of all triples (u, v, R), where $u, v \in \Gamma^*$ and $R \subseteq (G \times L^{|u|}) \times (G \times L^{|v|})$. We call the pair (u, v) the signature of (u, v, R). For technical reasons, we also include two special values: $\top$ for undefined value and 0 for neutral combine element.

The combine operation is the union of relations when the signatures are the same; otherwise $\top$ is returned. Formally,

$$\begin{array}{rcl} (u_0, u_1, R) \oplus (v_0, v_1, S) & = & \begin{cases} (u_0, u_1, R \cup S) & \text{if } u_0 = v_0 \text{ and } u_1 = v_1 \\ \top & \text{otherwise} \end{cases} \\ \top \oplus d & = & d \oplus \top \;=\; \top \quad \text{for all } d \in D \\ 0 \oplus d & = & d \oplus 0 \;=\; d \quad \text{for all } d \in D \;. \end{array}$$

The definition of the extend operation is more involved. For $l \in L$, we use l^n in the following to denote the sequence $l_1, \ldots, l_n$.

$$\begin{array}{rcl} (u_0, u_1, R) \otimes (v_0, v_1, S) & = & \begin{cases} (u_0, v_1 w, T_1) & \text{if } u_1 = v_0 w \text{ for some } w \in \Gamma^* \\ (u_0 w, v_1, T_2) & \text{if } v_0 = u_1 w \text{ for some } w \in \Gamma^* \\ \top & \text{otherwise} \end{cases} \\ \top \otimes d & = & d \otimes \top \;=\; \top \quad \text{for all } d \in D \text{ and } d \neq 0 \\ 0 \otimes d & = & d \otimes 0 \;=\; 0 \quad \text{for all } d \in D \;, \end{array}$$

where

$$T_1(g, l^{|u_0|}, g'', n^{|v_1|}, m^{|w|}) = \exists g', m^{|v_0|} : R(g, l^{|u_0|}, g', m^{|u_1|}) \wedge S(g', m^{|v_0|}, g'', n^{|v_1|})$$

$$T_2(g, l^{|u_0|}, m^{|w|}, g'', n^{|v_1|}) = \exists g', m^{|u_1|} : R(g, l^{|u_0|}, g', m^{|u_1|}) \wedge S(g', m^{|v_0|}, g'', n^{|v_1|})$$

The computation of T_1 (resp. T_2) can be seen as a join of R and S with respect to the variables that correspond to v_0 (resp. u_1)—the common elements of both signatures. The other elements remain unchanged, and are simply taken to the result. The neutral element 1 can be defined as $(\varepsilon, \varepsilon, \{(g, g) \mid g \in G\})$.

As an example, assume that the automaton on the right of Figure 4.11 has one global g and one local l. Its weights are partially listed as follows:

$$\begin{array}{rcl} l(p_0, a, q_{p_0,a}) & = & (a, a, (1, 0, 0, 1)) \\ l(q_{p_0,a}, b, q_0) & = & (a, ab, (1, y, 1, 0, 0)) \\ l(q_0, b, q_1) & = & (\varepsilon, ab, (x, x, y, y)) \;. \end{array}$$

Again, we use $x, y \in \{0, 1\}$ as a shorthand, and write e.g. $(1, y, 1, 0, 0)$ instead of a more awkward $\{(1, 0, 1, 0, 0), (1, 1, 1, 0, 0)\}$. The weight of the configuration $\langle p_0, abb \rangle$ is therefore

$$\begin{array}{rl} & (\varepsilon, ab, (x, x, y, y)) \otimes (a, ab, (1, y, 1, 0, 0)) \otimes (a, a, (1, 0, 0, 1)) \\ = & (\varepsilon, abb, (1, 1, 0, 0, y)) \otimes (a, a, (1, 0, 0, 1)) \\ = & (\varepsilon, abb, (1, 0, 1, 0, y)) \;, \end{array}$$

which implies that g has value 0, whereas l is 1, 0, y on the call stack abb, respectively.

4.3.2 Specialized reachability algorithms

We consider in the following some issues that arise when applying the reachability algorithms from Section 3.1 to the area of program testing. Both Algorithms 3.1 and 3.4 will be specialized for the pushdown models that are generated by the translator described in Section 4.2.

Quasi-one

We focus in this section on line 12 of Algorithm 3.1 where push rules are processed and new initial transitions with weight 1—the identity element—are produced. Observe that these initial transitions are only connected to transitions created in the same for-loop where weights are computed by $l(t) \otimes f(r)$ (line 13). Therefore, the only purpose that the identity element serves here is the fact that $l(t) \otimes f(r) \otimes 1 = l(t) \otimes f(r)$. Recall in the previous section that the identity element was defined as $(\varepsilon, \varepsilon, \{(g, g) \mid g \in G\})$. Obviously, the relation $\{(g, g) \mid g \in G\}$ may contain redundant pairs, i.e. pairs that are never used when joining with $l(t) \otimes f(r)$. This section explains how these pairs are eliminated and why this is useful.

Given a semiring $(D, \oplus, \otimes, 0, 1)$ and an element $d \in D$, we call $1_d \in D$ a *quasi-one of* d if and only if

$$d \otimes 1_d = d \ .$$

Intuitively, a quasi-one acts as the neutral element 1 for a specific element in the domain without requiring commutativity.

Following from the definition and the fact that all connected transitions have weights $l(t) \otimes f(r)$, we can replace the identity element in the algorithm with quasi-ones of $l(t) \otimes f(r)$ without effecting the correctness of the algorithm. If $l(t) \otimes f(r) = (\beta, \gamma'\gamma'', R)$, its quasi-one can be defined as (γ', γ', S), where

$$S(g', l', g'', l''') = \exists g, l, l'' : R(g, l, g', l', l'') \wedge g' = g'' \wedge l' = l''' \ .$$

It is obvious that $(\beta, \gamma'\gamma'', R) \otimes 1 = (\beta, \gamma'\gamma'', R) \otimes (\gamma', \gamma', S) = (\beta, \gamma'\gamma'', R)$.

The benefit of replacing the identity element with quasi-ones is that the quasi-ones represent global and local valuations when push rules are executed, i.e. when methods are called. Thus, the weight of an initial transition is always guaranteed to contain *only* valid global valuations and local valuations of the method on top of the stack. This makes it possible to obtain the actual valuations of globals and locals by inspecting only initial transitions. In the original algorithm (without quasi-ones) one must perform the extend operations on all transitions reachable from the initial transition to the final states in order to get the valid global and local valuations.

Dynamic thread creation

Recall that in Section 3.1.3, the reachability algorithm for pushdown networks (Algorithm 3.4) is initialized with a global configuration consisting of n threads. The algorithm makes use of the reachability algorithm for pushdown systems (Algorithm 3.1), where each thread is locally saturated without dynamic rules. However, to resemble many typical programming languages, including Java, it is more natural to start an analysis with a single main thread, and new threads can be constructed later during the analysis. The translator in Section 4.2 already produces dynamic rules when the bytecode instruction that forks new threads is encountered. The aim of this section is to give the formal definition of dynamic rules and extend the algorithms to support them.

We slightly modify the definition of pushdown networks in Section 2.2.3 to include dynamic rules. We no longer define a set of rules for each thread, but only a single set of rules that are shared among threads. Formally, a pushdown network is a triple $\mathcal{N} = (G, \Gamma, \Delta)$, where Δ is extended with rules of the form

$$\langle g, \gamma \rangle \hookrightarrow \langle g', \gamma' \rangle \rhd \gamma'' \ ,$$

where $g, g' \in G$ and $\gamma, \gamma', \gamma'' \in \Gamma$. Intuitively, the rule means that if the system's current global is g and a process has γ as the top-of-stack element, then the system can update the global to g', replace γ with γ', and create a new process with γ'' as the only symbol on the stack. A weighted pushdown network is similarly modified: $\mathcal{WN} = (\mathcal{N}, \mathcal{S}, f)$, where $\mathcal{N} = (G, \Gamma, \Delta)$ is a pushdown network, $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is a semiring, and $f : \Delta \to D$ is a function that assigns a weight from D to each rule in Δ. The global reachability relation between global configurations is extended for dynamic rules, i.e. if $\langle g, \gamma \rangle \overset{a}{\hookrightarrow} \langle g', \gamma' \rangle \rhd \gamma'' \in \Delta$, then:

$$\langle g, w_1, \ldots, \gamma w_i, \ldots, w_n \rangle \overset{a}{\Rrightarrow} \langle g', w_1, \ldots, \gamma' w_i, \ldots, w_n, \gamma'' \rangle \ ,$$

where $w_i \in \Gamma^*$ for all $i \in [n]$. Notice that the number of processes is increased by one on the right-hand configuration. However, as discussed in

Section 4.3.1, the relations between globals are not explicitly encoded in rules but implicitly represented in weights. Therefore, rules are only listed by their relations between stack symbols (cf. Section 4.2.1), e.g. $\gamma \hookrightarrow \gamma' \rhd \gamma''$.

What remains is to define weights for dynamic rules. Comparing to Table 4.1, a dynamic rule $\gamma \hookrightarrow \gamma' \rhd \gamma''$ should obviously have a relation of the form $G \times L \times G \times L \times L$. However, to conform to the semiring representation in Section 4.3.1, we need to introduce for every rule $\gamma \hookrightarrow \gamma' \rhd \gamma''$ a new stack symbol β' and split it into two rules:

$$\begin{array}{lcll} \gamma & \hookrightarrow & \beta' \rhd \gamma'' & (\gamma, \gamma'', R(g,l,g'',l'')) \\ \beta' & \hookrightarrow & \gamma' & (\gamma, \gamma', S(g,l,g',l')) \ , \end{array}$$

in which the first rule defines the relation R for the new thread, whereas the second rule gives the relation S for the current thread.

We now extend Algorithm 3.1 to handle dynamic rules when locally saturating a thread. We assume without loss of generality that the initial automaton that represents configurations of the main thread has one final state q_f. Moreover, we say that p is the only control location (cf. Section 4.2.1), and therefore the only initial state in the automaton. Let i' be the thread under consideration, i.e. the automaton $\mathcal{WA}_{i'} = ((Q_{i'}, \Gamma, \delta_{i'}, p, q_f), \mathcal{S}, l_{i'})$ is currently being saturated. We insert the following snippet after line 15 of Algorithm 3.1 (assuming that variables j, i, and *worklist* of Algorithm 3.4 are accessible).

```
16 forall $r = \gamma \hookrightarrow \beta' \rhd \gamma'' \in \Delta$ and $|T| < n$ do
17     create new $\overline{\mathcal{WA}_{i'}}$ and $\mathcal{WA}'$ (see text for definitions);
18     add $(j, i, (T, \mathcal{WA}')[\mathcal{WA}_{i'} \to \overline{\mathcal{WA}_{i'}}])$ to worklist;
```

The loop guard checks whether the size of the view tuple T is less than the threshold n. If true, two new automata $\overline{\mathcal{WA}_{i'}}$ and $\mathcal{WA}'$ are constructed. The automaton $\overline{\mathcal{WA}_{i'}}$ represents configurations of the thread i' after creating the new thread. Formally, $\overline{\mathcal{WA}_{i'}} = ((Q_{i'}, \Gamma, \overline{\delta_{i'}}, p, q_f), \mathcal{S}, \overline{l_{i'}})$, where

$$\overline{\delta_{i'}} = \{(p, \beta', q)\} \cup \{t' \in \delta_{i'} \mid t' \text{ reachable from } q\}$$

and

$$\overline{l_{i'}}(t) = \begin{cases} l_{i'}(p, \gamma, q) & \text{if } t = (p, \beta', q) \\ l_{i'}(t) & \text{if } t \in \{t' \in \delta_{i'} \mid t' \text{ reachable from } q\} \\ 0 & \text{otherwise.} \end{cases}$$

The weight of the new transition (p, β', q) is defined as a copy of the weight of the transition (p, γ, q). Therefore, the desired result, i.e. the weight after the new thread is forked, will be obtained at a later stage when processing (p, β', q) with the rule $\beta' \hookrightarrow \gamma'$.

On the other hand, the automaton $\mathcal{WA}'$ represents the configuration of the new thread. Formally, $\mathcal{WA}' = ((\{p, q_f\}, \Gamma, \{(p, \gamma'', q_f)\}, p, q_f), \mathcal{S}, l')$, where

$$l'(p, \gamma'', q_f) = l_{i'}(t) \otimes f(r) \ .$$

At line 18 we create a new view tuple $(T, \mathcal{WA}')[\mathcal{WA}_{i'} \to \overline{\mathcal{WA}_{i'}}]$ as a concatenation of T with $\mathcal{WA}'$ and replacing $\mathcal{WA}_{i'}$ with $\overline{\mathcal{WA}_{i'}}$. The new tuple is then added to *worklist* (of Algorithm 3.4) for future processing.

Eager splitting

We focus in this part on the split function in Algorithm 3.4. Recall that by using quasi-ones one can determine global valuations directly from weights of initial transitions. Thus, in eager splitting the function *split* only needs to take initial transitions into account, and splits on every possible global valuation in $G'(g') = \bigcup_{t \in \delta_{i'}} \{\exists l, g, l' : R(g, l, g', l') \mid l_{i'}(t) = (\gamma, \gamma', R)\}$.

$$\begin{aligned} split_{i'}(T) &= \{T_{i',g'} \mid g' \in G'(g')\}, \text{ where} \\ T_{i',g'} &= T[\mathcal{WA}_{i'} \to \mathcal{WA}_{i',g'}] \ . \end{aligned}$$

The new view tuple $T_{i',g'}$ is a copy of T with the i'-th view $\mathcal{WA}_{i'} = (\mathcal{A}_{i'}, \mathcal{S}, l_{i'})$ replaced by $\mathcal{WA}_{i',g'} = (\mathcal{A}_{i'}, \mathcal{S}, l_{i',g'})$. The configurations accepted by $\mathcal{WA}_{i',g'}$ are the configurations with global valuation g' that are accepted $\mathcal{WA}_{i'}$. Formally, for each $t \in \delta_{i'}$, we define the function

$$l_{i',g'}(t) = \begin{cases} l_{i'}(t) \wedge g' & \text{if } t \text{ is initial} \\ l_{i'}(t) & \text{otherwise.} \end{cases}$$

Lazy splitting

Recall that in Section 3.1.3 we make use of the *updated* relation $U'_{i'}(x, y)$ when computing the confluence relation $C_{i'}(x, y)$ after analyzing a local reachability for thread i'. The relation $U'_{i'}(x, y)$ defines the set of pairs of global valuations before and after the thread was active, hence the name "updated". As explained earlier, the pairs can be obtained by introducing another copy of globals. For this purpose, we extend the weight domain to include it, i.e. the weight domain now contains (u, v, R), where $u, v \in \Gamma^*$ and $R \subseteq (G \times G \times L^{|u|}) \times (G \times G \times L^{|v|})$. In contrast to the first copy, which stores the current global valuations, the second copy is used for memorizing the global valuations that the thread had when it became active.

We have the updated relation,

$$U'_{i'}(g'_0, g'_1) = \bigcup_{t \in \delta_{i'}} \{\exists g_0, g_1, l, l' : R(g_0, g_1, l, g'_0, g'_1, l') \mid l_{i'}(t) = (\gamma, \gamma', R)\} \ ,$$

to which Algorithm 3.5 can be directly applied for computing equivalence classes $G_1, \ldots, G_m$. The split function becomes

$$\begin{aligned} \mathit{split}_{i'}(T) &= \{T_{i',G_j} \mid G_j \in \{G_1, \ldots, G_m\}\}, \text{ where} \\ T_{i',G_j} &= (\mathcal{WA}_{1,G_j}, \ldots, \mathcal{WA}_{|T|,G_j}) \ . \end{aligned}$$

For all $k \in [|T|]$, the automaton $\mathcal{WA}_{k,G_j}$ is a copy of the k-th view of T, but with their global valuations updated by thread i'. Let $\mathcal{WA}_k = ((Q_k, \Gamma, \delta_k, p, q_f), \mathcal{S}, l_k)$ be the k-th view of T and $U'_{i',G_j}(g'_0, g'_1) = U'_{i'}(g'_0, g'_1) \wedge G_j(g'_0)$ be the updated relation with g'_0 restricted to G_j. We have

$$\begin{aligned} \mathcal{WA}_{k,G_j} &= ((Q_k, \Gamma, \delta_k, p, q_f), \mathcal{S}, l_{k,G_j}), \text{ and} \\ l_{k,G_j}(t) &= (u, v, R') \text{ for all } t \in \delta_k \text{ and } l_k(t) = (u, v, R) \ , \end{aligned}$$

where

$$R' = (\exists g'_0 : (\exists g'_1 : R(g_0, g_1, l^{|u|}, g'_0, g'_1, m^{|v|})) \wedge U'_{i',G_j}(g'_0, g'_1)) \wedge g'_0 = g'_1 \ .$$

Note that since the new copy of globals never change its valuations during the course of computing local reachability, the translator in Section 4.2 can be applied here without any modifications.

4.3.3 Counterexample extraction

Recall that given a Java method and a range of values to be tested, the translator in Section 4.2 produces a wrapper which calls the method with nondeterministic values inside the range. A reachability analysis (with Algorithm 3.1) always starts from the wrapper as the only method on the stack, i.e. the initial automaton $\mathcal{WA}$ accepts exactly one configuration $\langle p, \mathtt{w}_0 \rangle$, where

$\mathtt{w}_0$ is entry point of the wrapper. We say that a stack symbol (or, equivalently, a program point) γ is reachable, if it is possible reach a configuration with γ as the top symbol on the stack, i.e. $\langle p, \gamma w\rangle$ is reachable from $\langle p, \mathtt{w}_0\rangle$ for some $w \in \Gamma^*$. The fact that γ is reachable is immediately known during the analysis when a transition (p, γ, q) is added to $\mathcal{WA}_{post^*}$ for some q. However, it is unknown that *how* γ is reached. The aim of this section is to find an answer to this question.

In [51], an approach that constructs witness graphs during performing reachability analyses was presented. Basically, it keeps extra information "how transitions were added", and therefore can answer precisely how a configuration can be reached. We propose here another technique which extracts method arguments that the wrapper passed to the method under test (in order to reach a program point) from weights of transitions, i.e. from the information we already have. An obvious drawback of our approach is that it cannot directly generate any execution traces. We believe, however, that these traces can be easily obtained by using a debugging tool together with the extracted method arguments. Comparing to [51], our approach is simpler and requires no modifications of the *post** algorithm. Also, our approach requires much less memory as we extract the method arguments only when needed; unlike [51] where the witness graphs always need to be constructed.

Let us fix a weighted automaton returned by Algorithm 3.1 $\mathcal{WA}_{post^*} = (\mathcal{A}_{post^*}, \mathcal{S}, l)$, where $\mathcal{A}_{post^*} = (Q, \Gamma, \delta, p, q_f)$. Moreover, let t be a transition such that $l(t) = (a, w, R(g, l, g', m^{|w|}))$ for some $a \in \Gamma$, $w \in \Gamma^*$. We define the *witness* of t to be the relation $d(g, l) = \exists g', m^{|w|} : R(g, l, g', m^{|w|})$. Because of the way the weight is defined, it can be readily seen that the witness of t is the variable relation when the method that corresponds to t was called. Therefore, if $t = (p, \gamma, q)$ and there exists a transition from q to q_f, i.e. γ belongs to the method invoked by the wrapper, then the witness of t obviously contains method arguments that make γ reachable. Clearly, we also have to consider all paths from q to q_f (since they also make γ reachable). For this, we need to join the witness and the weights of the connected transitions together until reaching the final state.

Algorithm 4.1 realizes the idea above. The algorithm starts by initializing *workset* to the set of states that are connected to p. Then, it repeatedly removes a state from *workset*, finds adjacent states, and puts them into *workset* if their d-value changes. The function $d : Q \to G \times L$ implements the idea of witnesses, i.e. it maps "middle states" involved in reaching γ

```
**Input**: Weighted automaton $\mathcal{WA}_{post^*} = (\mathcal{A}_{post^*}, \mathcal{S}, l)$, where
        $\mathcal{A}_{post^*} = (Q, \Gamma, \delta, p, q_f)$, and $\gamma \in \Gamma$.
**Output**: The method arguments that lead to $\gamma$.
1  $workset := \{q \mid (p, \gamma, q) \in \delta\}$; $d := \lambda q.\emptyset$;
2  forall $t = (p, \gamma, q) \in workset$, where $l(t) = (a, \gamma, R)$ do
3      $d(q) := \exists g', l' : R(g, l, g', l')$;
4  while $workset \neq \emptyset$ do
5      remove $q$ from $workset$;
6      forall $t = (q, a, q') \in \delta$, where $l(t) = (b, ac, R)$ do
7          if $q' \neq q_f$ then
8              $v := d(q') \vee \exists g', l', l'' : (R(g, l, g', l', l'') \wedge d(q)(g', l'))$;
9              if $v \neq d(q')$ then
10                 $d(q') := v$;
11                 add $q'$ to $workset$;
12 return $\bigvee_{(q,a,q_f) \in \delta} d(q)$;
```

Algorithm 4.1: A counterexample generation algorithm

into variable valuations that can be used to reach γ from that states. For $t = (p, \gamma, q) \in \delta$, $d(q)$ is initialized to the arguments of the method of γ; formally $d(q)(g, l) = \exists g', l' : R(g, l, g', l')$ such that $l(t) = (a, \gamma, R)$ for some $a \in \Gamma$; otherwise, it is initialized to 0.

After removing a state q from the *workset*, the algorithm looks for all adjacent states q' (line 6). $d(q)$ already represents some method arguments if the method of γ was called when there is only one method on the call stack, i.e. when $q' = q_f$. Otherwise, we consider the transition $t = (q, a, q')$, and update the value of $d(q')$ at line 8 by joining $R(g, l, g', l', l'')$ such that $l(t) = (b, ac, R)$ with $d(q)(g', l')$ before abstracting g', l', l'' away. The state q' is added to *workset* if $d(q')$ is changed.

After *workset* is empty, the disjunction of $d(q)$ for all states q that corresponds to the call by the wrapper is returned.

A toy example

We demonstrate in this section the whole procedure of applying the reachability analyses in Chapter 3 to Java programs with a toy example. Consider the following method `m`:

```
static void m(int x) {
    if (x != 0) m(x + 2);
    return;
}
```

The method `m` has one parameter `x` and simply recursively calls itself with the argument `x + 2` if `x` is not equal to zero. Taking modulo arithmetic into account, one can see that `m` only returns when `x` is an even number; The method is compiled into a class file containing the following bytecode instructions:

```
m(I)V
 0: iload_0
 1: ifeq 10
 4: iload_0
 5: iconst_2
 6: iadd
 7: invokestatic m(I)V
10: return
```

The parameter `x` is represented by local variable 0. The first instruction pushes local variable 0 onto the operand stack. The second instruction (offset 1) pops the operand stack and compares it with zero. If true, the execution jumps to offset 10 and returns; otherwise it continues to the next instruction. The next instruction (offset 4) again pushes local variable 0. The instruction at offset 5 pushes a constant 2. The instruction `iadd` pops two elements from the operand stack, adds them, and pushes the result back. At offset 7, the method `m` is recursively called with the value on the operand stack as the argument.

We now translate the bytecode instructions into a weighted pushdown system. It can be seen that the method uses one local variable and two stack elements. As described in Section 4.2.3, we declare correspondingly four variables: local variable 0 ($\mathtt{lv}_0$), stack pointer (`sp`), and two stack elements ($\mathtt{s}[0]$ and $\mathtt{s}[1]$). The translation is straightforward as one instruction is translated into one pushdown rule (except the `ifeq` instruction) with a weight modeling the behavior of the instruction. The following weighted rules describe the resulting pushdown system for method m (with the default control location p omitted).

$$
\begin{array}{lll}
\mathtt{m}_0 & \hookrightarrow & \mathtt{m}_1 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{m}_1 & \hookrightarrow & \mathtt{m}_{10} & \mathtt{s}[\mathtt{sp} - 1] = 0 \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{m}_1 & \hookrightarrow & \mathtt{m}_4 & \mathtt{s}[\mathtt{sp} - 1] \neq 0 \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{m}_4 & \hookrightarrow & \mathtt{m}_5 & \mathtt{s}'[\mathtt{sp}] = \mathtt{lv}_0 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{m}_5 & \hookrightarrow & \mathtt{m}_6 & \mathtt{s}'[\mathtt{sp}] = 2 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{m}_6 & \hookrightarrow & \mathtt{m}_7 & \mathtt{s}'[\mathtt{sp} - 2] = \mathtt{s}[\mathtt{sp} - 2] + \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = \mathtt{sp} - 1 \\
\mathtt{m}_7 & \hookrightarrow & \mathtt{m}_0\ \mathtt{m}_{10} & \mathtt{lv}_0' = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = 0 \wedge \mathtt{sp}'' = \mathtt{sp} - 1 \\
\mathtt{m}_{10} & \hookrightarrow & \varepsilon &
\end{array}
$$

To test the method m, we assume that every variable has two bits, i.e. by using a two's-complement system, its value can only be inside the range $[-2, 1]$ We construct a wrapper that calls m with all possible values inside the range.

$$
\begin{array}{lll}
\mathtt{w}_0 & \hookrightarrow & \mathtt{w}_1 & -2 \leq \mathtt{s}[\mathtt{sp}] \leq 1 \wedge \mathtt{sp}' = \mathtt{sp} + 1 \\
\mathtt{w}_1 & \hookrightarrow & \mathtt{m}_0\ \mathtt{w}_2 & \mathtt{lv}_0' = \mathtt{s}[\mathtt{sp} - 1] \wedge \mathtt{sp}' = 0 \wedge \mathtt{sp}'' = \mathtt{sp} - 1
\end{array}
$$

We have now defined the weighted pushdown system $\mathcal{WP} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (\{p\}, \Gamma, \Delta)$ such that Γ, Δ, and f are implicitly defined above.

Next, we define the weight domain D of the semiring $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$. See Section 4.3.1 for the definitions of $\oplus$, $\otimes$, 0, and 1. With four variables, each has two bits, we have $L = \{0, 1\}^8$. We define the following meanings to words in L: the first two bits are for local variable 0, followed by two bits for the stack pointer, and two bits for each stack element 0 and 1, respectively. Let $I = \{-2, -1, 0, 1\}$ be the set of all possible values. In the following, we informally use a quintuple to represent a subset of L, and liberally mix I and variables as a shorthand. For instance, given a variable a that can take any value inside the range, the tuple $(a, 1, a, I)$ represents $\{a01ab \mid a, b \in \{0, 1\}^2\}$. Similarly, a concatenation of tuples represents a relation over L.

We now define the weighted automaton $\mathcal{WA} = (\mathcal{A}, \mathcal{S}, l_0)$, where $\mathcal{A} = (\{p, q\}, \Gamma, \delta_0, \{p\}, \{q\})$. The set δ_0 contains only one transition (p, w_0, q) with weight $(\varepsilon, \mathtt{w}_0, (0, 0, I, I))$. Therefore, $L(\mathcal{WA}) = \{\langle p, \mathtt{w}_0\rangle\}$. In other words, this means that we start the analysis at the wrapper method, where its local variable 0 and the stack pointer are set to zero. The stack contents are not initialized.

What remains is to apply Algorithm 3.1 to $\mathcal{WP}$ and $\mathcal{WA}$. This produces $\mathcal{WA}_{post^*} = (\mathcal{A}_{post^*}, \mathcal{S}, l)$, where $\mathcal{A} = (Q, \Gamma, \delta, \{p\}, \{q\})$. Because of the push rules, we have the $Q = Q_0 \cup \{q_{p,\mathtt{m}_0}\}$. The set of transitions δ and the weight function l are first initialized to δ_0 and l_0, respectively. Then, the algorithm sequentially adds new transitions and their corresponding weights as described in Table 4.3.

Notice that at step 2, the weight of transition $(p, \mathtt{m}_0, q_{p,\mathtt{m}_0})$ is the quasi-one of the weight of $(q_{p,\mathtt{m}_0}, \mathtt{w}_2, q)$. At step 10, however, the weight of transition $(p, \mathtt{m}_0, q_{p,\mathtt{m}_0})$ is not modified (thus no need to consider the transition again in the algorithm), because it already includes the quasi-one of the weight of $(q_{p,\mathtt{m}_0}, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$. Also, because of the use of the quasi-ones, one can inspect the weight of $(p, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$ (step 11) and conclude that the valuations of local variable 0 at $\mathtt{m}_{10}$ are either 0 or -2 for all configurations having $\mathtt{m}_{10}$ on top of the stack. Recall that $\mathtt{m}_{10}$ corresponds to the `return` statement in the Java version of method `m`. This means the `return` statement is only reachable when the parameter `x` is either 0 or -2, i.e. the even numbers inside the range.

Notice also that although the method under test does not terminate with certain argument values, our analysis always terminates, and is able to find out the argument values that make the method terminates.

#	Transitions	Extended with	Results	
1	$(p, \mathtt{w}_0, q)$	$\mathtt{w}_0 \hookrightarrow \mathtt{w}_1$	$(p, \mathtt{w}_1, q)$	$(\varepsilon, \mathtt{w}_1, (0, 1, a, I))$
2	$(p, \mathtt{w}_1, q)$	$\mathtt{w}_1 \hookrightarrow \mathtt{m}_0\ \mathtt{w}_2$	$(q_{p,\mathtt{m}_0}, \mathtt{w}_2, q)$	$(\varepsilon, \mathtt{m}_0\mathtt{w}_2, (a, 0, b, c)(0, 0, a, I))$
			$(p, \mathtt{m}_0, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_0, (a, 0, b, c)(a, 0, b, c))$
3	$(p, \mathtt{m}_0, q_{p,\mathtt{m}_0})$	$\mathtt{m}_0 \hookrightarrow \mathtt{m}_1$	$(p, \mathtt{m}_1, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_1, (a, 0, b, c)(a, 1, a, c))$
4	$(p, \mathtt{m}_1, q_{p,\mathtt{m}_0})$	$\mathtt{m}_1 \hookrightarrow \mathtt{m}_{10}$	$(p, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_{10}, (0, 0, b, c)(0, 0, 0, c))$
		$\mathtt{m}_1 \hookrightarrow \mathtt{m}_4$	$(p, \mathtt{m}_4, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_4, (u, 0, b, c)(u, 0, u, c))$
5	$(p, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$\mathtt{m}_{10} \hookrightarrow \varepsilon$	$(p, \varepsilon, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \varepsilon, (0, 0, b, c))$
6	$(q_{p,\mathtt{m}_0}, \mathtt{w}_2, q)$	$(p, \varepsilon, q_{p,\mathtt{m}_0})$	$(p, \mathtt{w}_2, q)$	$(\varepsilon, \mathtt{w}_2, (0, 0, 0, I))$
7	$(p, \mathtt{m}_4, q_{p,\mathtt{m}_0})$	$\mathtt{m}_4 \hookrightarrow \mathtt{m}_5$	$(p, \mathtt{m}_5, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_5, (u, 0, b, c)(u, 1, u, c))$
8	$(p, \mathtt{m}_5, q_{p,\mathtt{m}_0})$	$\mathtt{m}_5 \hookrightarrow \mathtt{m}_6$	$(p, \mathtt{m}_6, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_6, (u, 0, b, c)(u, 2, u, 2))$
9	$(p, \mathtt{m}_6, q_{p,\mathtt{m}_0})$	$\mathtt{m}_6 \hookrightarrow \mathtt{m}_7$	$(p, \mathtt{m}_7, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_7, (u, 0, b, c)(u, 1, u + 2, 2))$
10	$(p, \mathtt{m}_7, q_{p,\mathtt{m}_0})$	$\mathtt{m}_7 \hookrightarrow \mathtt{m}_0\ \mathtt{m}_{10}$	$(q_{p,\mathtt{m}_0}, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_0\mathtt{m}_{10}, (u, 0, b, c)(u + 2, 0, d, e)(u, 0, u + 2, 2))$
11	$(q_{p,\mathtt{m}_0}, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$(p, \varepsilon, q_{p,\mathtt{m}_0})$	$(p, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \mathtt{m}_{10}, (-2, 0, b, c)(-2, 0, 0, 2))$
				$\oplus\ (\mathtt{m}_0, \mathtt{m}_{10}, (0, 0, b, c)(0, 0, 0, c))$
12	$(p, \mathtt{m}_{10}, q_{p,\mathtt{m}_0})$	$\mathtt{m}_{10} \hookrightarrow \varepsilon$	$(p, \varepsilon, q_{p,\mathtt{m}_0})$	$(\mathtt{m}_0, \varepsilon, (\{-2, 0\}, 0, b, c))$
13	$(q_{p,\mathtt{m}_0}, \mathtt{w}_2, q)$	$(p, \varepsilon, q_{p,\mathtt{m}_0})$	$(p, \mathtt{w}_2, q)$	$(\varepsilon, \mathtt{w}_2, (0, 0, \{-2, 0\}, I))$

Table 4.3: Computational steps. Variables can take any values from I, except u, which must be nonzero.

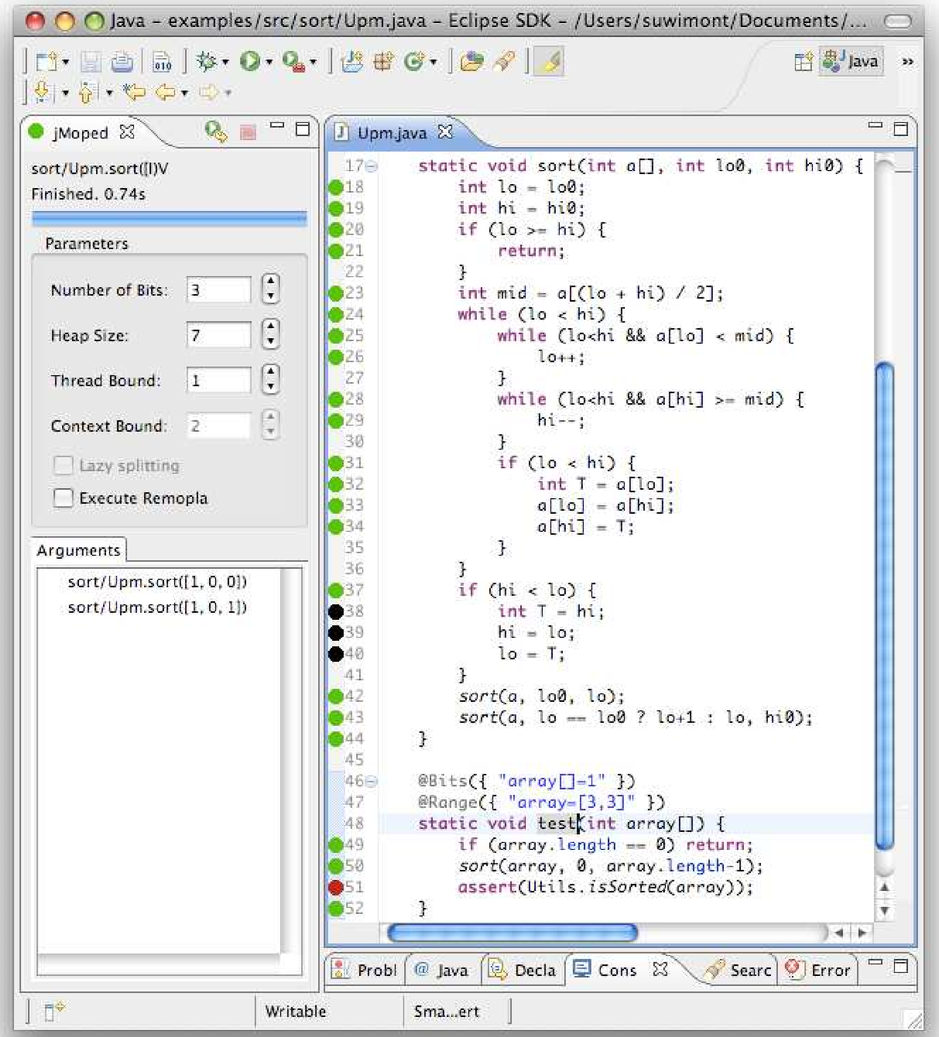
Chapter 5

Experiments with jMoped

This chapter presents a tool named jMoped, which implements the reachability algorithms and the translator discussed in Section 3.1 and 4.2, and reports on experimental results. jMoped is an Eclipse plug-in which enables Java developers to easily test their programs without knowing the model-checking techniques behind it. Figure 5.1 shows a screenshot when running with a quicksort implementation taken from [46]. The left-hand side is the plug-in interface. The right-hand side shows the code and the analysis results. To test a program, users simply select a method where the analysis should start. In the example, the method `test` starting at line 48 was chosen.

One can think of jMoped as a virtual machine that can execute the code for all possible parameter values (within given bounds) in a single run, in contrast to the Java virtual machine which always executes the program with a concrete value. During the analysis, jMoped graphically displays its progress. First, black markers are placed in front of all statements that are statically reachable from the selected method. While the checker is running, the parts of the state space found to be reachable are mapped back to the Java program, and the appearance of the corresponding markers is changed. When a black marker turns green, it means that the corresponding Java statement is reachable from *some* argument values. A red marker means that an assertion statement has been violated by some argument values. Other markers indicate null pointer exceptions, array bound violations, and heap overflows. See the tool's website [34] for more information.

If an assertion is violated, users can generate *all* argument values that violate the assertion. JUnit test cases can also be generated for future testing. An example of the argument values can be seen in lower left part of Figure 5.1,


Java - examples/src/sort/Upm.java - Eclipse SDK - /Users/suwimont/Documents/...
Java
jMoped
Upm.java
Probl
Java
Decla
Cons
Searc
Error
Writable
Sma...ert

sort/Upm.sort([I)V
Finished. 0.74s

Parameters

Number of Bits: 3
Heap Size: 7
Thread Bound: 1
Context Bound: 2
Lazy splitting
Execute Remopla

Arguments

sort/Upm.sort([1, 0, 0])
sort/Upm.sort([1, 0, 1])

```java
static void sort(int a[], int lo0, int hi0) {
    int lo = lo0;
    int hi = hi0;
    if (lo >= hi) {
        return;
    }
    int mid = a[(lo + hi) / 2];
    while (lo < hi) {
        while (lo<hi && a[lo] < mid) {
            lo++;
        }
        while (lo<hi && a[hi] >= mid) {
            hi--;
        }
        if (lo < hi) {
            int T = a[lo];
            a[lo] = a[hi];
            a[hi] = T;
        }
    }
    if (hi < lo) {
        int T = hi;
        hi = lo;
        lo = T;
    }
    sort(a, lo0, lo);
    sort(a, lo == lo0 ? lo+1 : lo, hi0);
}

@Bits({ "array[]=1" })
@Range({ "array=[3,3]" })
static void test(int array[]) {
    if (array.length == 0) return;
    sort(array, 0, array.length-1);
    assert(Utils.isSorted(array));
}
```

Figure 5.1: A view of the plug-in.

where the assertion was violated when the method `sort` was called with the arrays `[1,0,0]` and `[1,0,1]`, i.e. the quicksort implementation does not sort these two arrays correctly. The assertion makes use of the method `Utils.isSorted(array)`, which returns true if the `array` is sorted, or false otherwise. Its code is not shown here.

There are two important parameters to jMoped required from users when testing a sequential program—the number of *bits* of the program variables and the *heap size*. Section 4.2 discusses the meanings of these two parameters in great detail. However, as users it suffices to know the following: The number of bits restricts the range of every number that appears in the program, i.e. with b bits, every integer must be in the range $[-2^{b-1}, 2^{b-1}-1]$ (with an exception when $b = 1$, the range is $[0, 1]$). This includes for instance constants, integer variables, and array lengths. Like in Java, jMoped implements modulo arithmetic, and therefore it is possible to obtain e.g. a negative number by adding two positive numbers. The heap size directly affects the number of objects that can be instantiated. jMoped simulates the heap when manipulating objects, i.e. when an object is created, it occupies a part of the heap whose size depends on the size of the object. The size of an object, on the other hand, depends on the number of instance fields it has. The analysis in Figure 5.1 was performed with 3 bits and heap size 7. It is also possible to specify numbers of bits for individual variables, parameters, or fields by using annotations. The annotations at lines 46–47 in Figure 5.1 indicate that the length of `array` is three, and each of its element has one bit, i.e. elements can only be either 0 or 1.

Given an argument range of a test method, one might argue that it is also possible to test the method simply by executing the test method for each possible argument value consecutively. This feature is also implemented in jMoped. The idea is that the translator will generate a wrapper (cf. Section 4.2.3) that repeatedly calls the test method with different argument values inside the range. Then, the translated pushdown model will be *executed* by a virtual machine in a similar way to the Java virtual machine executing bytecode instructions. Users can turn this mode on by checking the box “Execute Remopla” in the plug-in. (Remopla is the name of the language that we use for representing pushdown systems.)

Two more parameters are involved when testing multithreaded programs: *thread bound* and *context bound*. The thread bound limits the number threads in the program. No new threads are forked when the bound is reached. The context bound limits the number times threads can become active. Context bound k indicates that threads can be active at most k times, i.e. they can communicate at most $k-1$ times. See Section 3.1.3 and 4.3.2 for detailed discussions.

For more details of the tool, see [34]. This chapter reports on experimental results with several examples. All experiments were performed on an AMD 3 GHz machine with 64 GB memory.

5.1 BDDs vs. bit vectors

In Section 4.3.1, we discussed a representation of variable relations as semirings. We, however, did not specify any concrete data structure that can be used to store these relations. This section focuses on this issue, and discusses advantages and disadvantages of different data structures.

jMoped has two implementations for variable relations: bit vectors and BDDs. As the name suggests, in bit vectors a relation is explicitly stored as a set of sequences of numbers. Sets are implemented by using a hash function to ensure that same sequences are kept only once. Operations on relations such as composition involve considering each element in the relations one-by-one. In contrast, with BDDs (see Section 2.3) we try to reduce space and time required for storing and performing operations on relations. jMoped uses the JavaBDD library [61] for manipulating BDDs. JavaBDD includes a pure Java implementation as well as interfaces to several C libraries. All experiments in this thesis were performed with the interface to the CUDD library [55].

The method `test1` in Figure 5.2 exemplifies the first difference between the two implementations. When analyzing the method with jMoped, it first constructs a wrapper method, which calls `test1` with `x` having all possible values within the range $[-512, 511]$, if the number of bits is set to 10. Clearly, with bit vectors all 1024 values must be explicitly stored using 1024 vectors, and each of them must be explicitly updated each time `y` is increased in the loop. The loop is repeated 100 times, so 102400 vectors must be updated in total. jMoped needs 304.2 seconds to test the method in this case. On the other hand, when using BDDs we hope for a more compact data structure, on which operations can be performed more efficiently. Here, all 1024 values can be represented in a single BDD. Operations are much cheaper as a result, and jMoped only requires 7.2 seconds in this case.

Nevertheless, BDDs are not a silver bullet for all types of relations. Notice

```
public class O {
    public static void test1(int x) {
        int y = x;
        for (int i = 0; i < 100; i++)
            y++;
        assert(x + 100 == y);
    }

    int x;
    public O(int x) {
        this.x = x;
    }
    public static void test2() {
        O o = null;
        for (int i = 0; i < 100; i++)
            o = new O(i);
        assert(o.x == 99);
    }
}
```

Figure 5.2: Two test methods for comparing bit vectors with BDDs.

that the method `test1` does not make use of the heap. We experienced slowdowns with BDDs when heaps are large. Consider for instance the method `test2` in Figure 5.2, which repeatedly allocates 100 objects of type `O` into the heap. The heap is initially empty, and contains one more object each time the loop is executed. Therefore, when analyzing the method the variable relation of the statement inside the loop must represent the union of the empty heap, the heap with one object, the heap with two objects, and so on. The corresponding BDD becomes much more complex than the explicit representation with bit vectors. In the experiment, more than 215000 BDD nodes are required for the relation inside the loop when variables have 10 bits. jMoped requires 122.8 seconds to test the method with BDDs, but only needs 2.5 seconds with bit vectors.

To summarize from experience, BDDs tend to perform better when testing ranges are large. This makes enumerating all possible valuations with bit vectors becomes too slow or even impossible. BDDs benefit from the fact that they can "compress" those valuations into smaller representations, on which operations can be performed efficiently. On the other hand, heap manipulations are more expensive on BDDs. The difference on this issue becomes more visible when programs under test require large amounts of heaps, or repeatedly modify heaps, e.g. in loops.

5.2 Quicksort

Quicksort is a sorting algorithm based on the divide-and-conquer paradigm. Given an array to be sorted, the algorithm takes the following steps:

1. An element of the array is picked as the *pivot*.

2. The array is reordered such that all elements less than or equal to pivot are moved to the left of pivot. The elements that are greater than pivot are moved to the right.

3. The array is divided at the pivot, and the two sub-arrays are solved recursively.

We consider in this section two different versions of quicksort implementations. In both implementations, given an array of integer values, the returned array should be sorted in nondescending order. We use jMoped to test this property with various array lengths, and report on experimental results. All experiments were performed in the BDD mode. The reason is justified, since the bit-vector mode tends to be slower when testing ranges are large, i.e. when arrays under test contains nontrivial numbers of elements. See Section 5.1 for discussions on this issue.

5.2.1 Version 1

The first version, taken from [46], is shown in Figure 5.3. The method `sort` is supposed to sort the array `a` from the index `lo0` to `hi0`. We use the following method to test whether, given an array `a`, `a` is always correctly sorted after the method returns.

```
static void test(int a[]) {
    sort(a, 0, a.length - 1);
    assert(isSorted(a));
}
```

When analyzed with jMoped, the `test` method is wrapped by a new method that creates the array `a` with nondeterministic values (within a given range). The wrapper then calls the test method with `a` as the argument. We ensure that `a` is correctly sorted after the call to `sort` by inserting an assertion statement in the test method. The method `isSorted(a)` (code not shown) returns true if `a` is sorted, or false otherwise.

The first half of Table 5.1 shows experimental results: time and numbers of BDD nodes required, when running jMoped on the test method with different array lengths. We assume in all experiments that every array element has only one bit, i.e. its value can be either 0 or 1. jMoped found assertion violations when the array lengths are greater than two, which indicates that there are some arrays that are not correctly sorted. Table 5.1 also lists the numbers of such arrays.

Notice that until now what jMoped does is simply test the method with all possible arrays of given lengths. Obviously, one can argue that the same effect can be achieved by running the test method one by one for each possible array. For this, we use the option “Execute Remopla” to execute the underlying pushdown system for each array. The experimental results are listed with the label “Exec. time” in Table 5.1. One can see that the execution time rapidly worsens when array lengths grow.

It is well known that quicksort can take quadratic time when pivots are poorly selected, resulting in partitions that are extremely unequal. A safe so-

```
static void sort(int a[], int lo0, int hi0) {
    int lo = lo0;
    int hi = hi0;
    if (lo >= hi) {
        return;
    }
    int mid = a[(lo + hi) / 2];
    while (lo < hi) {
        while (lo<hi && a[lo] < mid) {
            lo++;
        }
        while (lo<hi && a[hi] >= mid) {
            hi--;
        }
        if (lo < hi) {
            int T = a[lo];
            a[lo] = a[hi];
            a[hi] = T;
        }
    }
    if (hi < lo) {
        int T = hi;
        hi = lo;
        lo = T;
    }
    sort(a, lo0, lo);
    sort(a, lo == lo0 ? lo+1 : lo, hi0);
}
```

Figure 5.3: Quicksort version 1

Lengths		3	6	9	12	15	18
Original	Time (s)	0.9	1.1	3.5	11.3	36.3	159.2
	Nodes ($\times 10^6$)	0.03	0.2	0.5	1.2	2.8	7.5
	Unsorted	2	20	216	1456	14032	93600
	Exec. time (s)	0.7	0.9	1.2	9.8	93.4	973.3
Random	Time (s)	0.9	1.5	7.5	63.5	525.8	4136.7
	Nodes ($\times 10^6$)	0.03	0.3	1.0	4.6	18.8	105.4
	Unsorted	4	57	502	4083	32752	262125
	Exec. time (s)	1.0	1.0	1.3	9.7	94.9	1036.3
	Unsorted	2 (50%)	18 (32%)	150 (30%)	1311 (32%)	10525 (32%)	85318 (33%)

Table 5.1: Experimental results: quicksort version 1

lution is to pick pivots randomly, and relies on the unlikeliness that random pivots would consistently lead to poor partitions. Notice that the implementation in Figure 5.1 always picks the pivots in the middle of arrays. In the following, we consider an alternative version by modifying the code in Figure 5.1 so that `mid` is randomly selected from `a` (between `lo` and `hi`).

Nevertheless, adding randomness to an algorithm makes it more difficult to test, because it is no longer possible to simply enumerate all possible input arguments to ensure the correctness. In our example, picking some pivots will not correctly sort an array, while some other pivots will do. As a result, we might end up with successful tests just because pivots we picked correctly sort the arrays under test. To cope with the problem, jMoped models the random function as a function that always returns a nondeterministic value. Therefore, when no errors can be found with jMoped, one is certain that the algorithm always works correctly for *any* random values.

We reran the experiments again, but this time with random pivots. The results are shown in the second half of Table 5.1. One can see that, compared to the original version, more time is required for the same array lengths as more degrees of nondeterminism are involved. Again, we compare the results with the time required to execute the underlying pushdown system. The execution time remains mostly unchanged when comparing to the time for the original version. However, only parts of assertion violations were found. We ran each experiment three times and computed the average numbers of unsorted arrays. The numbers in parentheses are percentages of errors found compared to the total numbers of errors.

5.2.2 Version 2

Figure 5.4 lists another version of quicksort, taken from [60] with a slight modification. It sorts the array `a` from index `left` to index `right`. This implementation is carefully optimized in many aspects. It calls `insertionsort`, another sorting algorithm (code not shown), when the array to be sorted is small enough, i.e. when the difference between `right` and `left` is less than or equal to a predefined constant `CUTOFF`.

Pivots are selected by calls to the method `median3`. Given an array `a` and indices `left` and `right`, the method `median3` works as follows: (i) it compares the elements of `a` at `left`, `right`, and `center` = (`left` + `right`)/2 before putting the least element to `a[left]`, the median to `a[center]`, and the largest element to `a[right]`; (ii) as another side effect, the median is swapped with the second last element, i.e. the array elements `a[center]` and `a`[`right` − 1] are swapped; and (iii) the median is returned as the pivot.

The benefit of implementing `median3` this way is twofold. First, the partitioning can start at `i` = `left`+1 and `j` = `right`−2 because the elements at `left` and `right` are already in their proper partitions. Moreover, since `a[left]` is smaller than the pivot, `j` will never run past the end, thus saving a check for the array bound. The same applies to `i`, since the partitioning stops on elements larger than or equal to the pivot, so it always stops at `right` − 1.

We use jMoped to test the method `sort` in a similar way to the previous experiment such that we call `sort` with arrays of various lengths, and test whether the returned arrays are correctly sorted. The constant `CUTOFF` is set to 3. The time and number of BDD nodes required for each array length are listed Table 5.2. Again, all array elements are limited to one bit, i.e. their values can be either 0 or 1. No errors were found.

For comparisons, Table 5.2 also includes execution time where the method `sort` is tested one by one for each array within a given range. One can see that the execution time increases rapidly as array lengths grow, and it takes more than two hours when the array under test is of length 24.

As pointed out in [60], it is tempting to change the initializations of `i` and `j` and the inner loops to the following:

```
int i = left + 1; j = right - 2;
```

```
private static int median3(int[] a, int left, int right) {
    int center = (left + right)/2;
    if (a[center] < a[left])   swap(a, center, left);
    if (a[right]  < a[left])   swap(a, right,  left);
    if (a[right]  < a[center]) swap(a, right,  center);

    swap(a, center, right - 1);  // Pivot at right - 1
    return a[right - 1];
}

static void sort(int[] a, int left, int right) {
    if (left + CUTOFF >= right) {
        insertionsort(a, left, right);
        return;
    }
    int pivot = median3(a, left, right);
    int i = left, j = right - 1;
    while (true) {
        while (a[++i] < pivot) {}
        while (a[--j] > pivot) {}
        if (i < j) swap(a, i, j);
        else break;
    }
    swap(a, i, right - 1);  // Restores pivot
    sort(a, left, i - 1);
    sort(a, i + 1, right);
}
```

Figure 5.4: Quicksort version 2

Lengths	3	6	9	12	15	18	21	24
Time (s)	0.8	1.1	4.0	19.5	98.1	397.9	1220.7	8402.3
Nodes ($\times 10^6$)	0.02	0.2	0.7	2.4	8.7	31.4	116.7	479.4
Exec. time (s)	0.8	1.5	1.7	8.1	74.8	731.1	6492.3	timeout

Table 5.2: Experimental results: quicksort version 2

```
while (true) {
    while (a[i] < pivot) i++;
    while (a[j] > pivot) j--;
    if (i < j) swap(a, i, j);
    else break;
}
```

However, the modification not work because of the infinite loop when `a[i]` and `a[j]` are equal to `pivot`. We reran the analyses again with the buggy code for each array length listed in Table 5.2. jMoped was able to find out almost immediately that the method `sort` never return with any arrays having lengths not less than 9, and only half of the possible arrays with length 6 are sorted correctly, whereas the other half are stuck inside the loop.

Notice that classical testing is not applicable here because of the infinite loop. The experiment exemplifies another use of jMoped in that it always terminates even when the code under test does not. In this respect, jMoped is more beneficial than classical testing when it is not possible to differentiate between programs that are not terminate from others that simply take long time. This error can be captured with jMoped by comparing the numbers of possible argument values before and after a call to method under test. Usually, they should be equal, indicating that the method always terminates with every possible argument value.

5.3 jMoped BDD library

This experiment focuses on applying jMoped to a part of itself. The purpose is to demonstrate its usability for code closer to real applications.

As discussed in Chapter 4, jMoped translates bytecode instructions into weighted pushdown models, where weights are relations between variables. Let R be a translated relation. When using BDDs for encoding weights, it is possible to translate R into a BDD before performing reachability analyses. However, an obvious drawback is that the BDD for R can be very large, especially when variables have non-trivial numbers of bits. For instance, if R is $x' = x + 1$, where x is an integer variable with n bits, the BDD for R must contain the relation $\{(x, x+1) \mid x \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}\}$, which grows exponentially in n. This makes the reachability analyses become impractical very quickly (even before the analyses start) when the number of bits grows.

To cope with the problem, jMoped only build BDDs when they are needed, i.e. when performing extend operations (see Section 3.1). The idea is that given a BDD A and a relation R, jMoped first inspects all possible values in A of variables that appear in R. Then, it constructs a new BDD B satisfying the relation R, but containing only inspected values from A. One can see that the resulting BDDs can be much smaller, especially when numbers of bits are large but only a handful of values appear in A. Continuing with the above example, assume that the values of x in A can only be $\{0, 1\}$, jMoped then only constructs B representing the relation $\{(0, 1), (1, 2)\}$, where the pairs represent the relation between x and x', respectively.

For this purpose, we have implemented a library for jMoped that manipulates BDDs for different types of relations. Given a BDD and a relation written in a text format (e.g. $x' = x + 1$), the library returns a new BDD, which is the result of extending the BDD with the relation. The library supports all relations generated from the translator. Recall from Section 4.2 that some relations have similar behaviors, and therefore can be grouped together. For example, given a constant x, the group Push pushes x onto the operand stack. As a result, the library was constructed in such a way that it contains functions that handle each relation type. We use jMoped to test these functions in this experiment, e.g. to test the group Push, we create an initial BDD, set the constant x to be nondeterministic, and check whether the BDD that the library returns correctly has x on top of the stack.

Figure 5.5 gives a glimpse of the function Load. It takes a BDD `bdd` and a local variable's `index`, and loads the local variable at index onto the operand stack. The method first obtains BDD domains—a set of BDD variables—of the stack pointer, the stack element pointed by the stack pointer, and the local variable via external methods (code not shown). Then, old values of the stack pointer and the stack element are abstracted away before updating with new values from the local variable. The method `buildEquals` constructs a BDD representing the equality between BDD variables of two BDD domains.

Table 5.3 summarizes the experiments. Each experiment tests whether the library works correctly within a bounded nondeterministic input range. It first creates a BDD and a relation with variables of interest having values inside the range, inputs them into the library, then checks whether the output BDD is correctly constructed, i.e. whether the values are updated and stored in expected variables. The translated pushdown system for the library contains approximately 145000 rules. Table 5.4 shows the experimental results when the variables under test have 1 bit. The table lists time

```
/**
 * Reads a BDD encoding a variable relation, and
 * loads a local variable onto the operand stack.
 * @param bdd the BDD.
 * @param index the local variable's index.
 * @return a new BDD, loaded with the local variable at index.
 */
private BDD load(BDD bdd, int index) {
    /*
     * Gets the BDD domains of the stack pointer, the stack
     * element pointed by the stack pointer, and the local
     * variable at index.
     */
    BDDDomain spdom = getStackPointerDomain();
    int sp = bdd.scanVar(spdom).intValue();
    BDDDomain s0dom = getStackDomain(sp);
    BDDDomain vdom = getLocalVarDomain(index);

    /*
     * Abstracts the stack pointer and
     * the stack element from the BDD.
     */
    BDDVarSet varset = spdom.set().unionWith(s0dom.set());
    BDD newbdd = bdd.exist(varset);
    varset.free();

    // Updates the stack pointer and the stack element.
    newbdd.andWith(spdom.ithVar(sp + 1));
    newbdd.andWith(s0dom.buildEquals(vdom));
    return newbdd;
}
```

Figure 5.5: A simplified version of a method in the jMoped's BDD library.

and heaps required to test each relation type. One can see that the time is greatly influenced by the heap sizes used, especially for complex relations that manipulate heaps such as Arrayload and Arraystore (cf. Section 4.2). All experiments were performed with weights encoded as bit vectors.

Note that tests are fully automatic, and the library was tested as is, i.e. neither preprocessing nor modifications were performed. However, since jMoped cannot afford very large heaps, in order to make the tests possible we have written a simple library that performs basic operations on BDDs. The library caches less than JavaBDD—the library jMoped actually uses. It is slower, but requires less memory. This results in heap sizes that are small enough for jMoped.

5.4 `java.util.Vector` class

The rest of this chapter deals with multithreading programs. In this experiment we consider the class `java.util.Vector` from the Java library. The `Vector` class implements a growable array of objects. In [59], a race condition in a constructor of `Vector` was reported. The following test method illustrates the situation where the race condition can occur.

```
static void test(Integer x) {
    final Vector<Integer> v1 = new Vector<Integer>();
    v1.add(x);
    new Thread(new Runnable() {
        public void run() { v1.removeAllElements(); }
    }).start();
    Vector<Integer> v2 = new Vector<Integer>(v1);
    assert(v2.isEmpty() || v2.elementAt(0) == x);
}
```

The method creates two vectors. First an empty vector `v1` is created, and then an integer `x` is added to it as its first element. After that, a new thread is forked, which removes all elements from `v1` (only `x` in this case). In parallel, the first thread creates a copy `v2` of `v1`. Intuitively, only two cases are possible: if the elements of `v1` are removed before `v2` is created, then `v2` is empty; if `v2` is created before the elements of `v1` are removed, then the first element of `v2` is equal to `x`. The last line of code asserts this property.

Types	Variables	Tests the output BDDs whether ...
Push	constant	constant is pushed.
Load	local var.	local variable is pushed.
Store	stack	stack is popped to local variable.
Globalload	global var.	global variable is pushed.
Globalstore	stack	stack is popped to global variable.
Unary	stack	stack is modified w.r.t. unary operation.
Binary	stack	stack is modified w.r.t. binary operation.
Inc	local var.	local variable is incremented.
New	class id	class id stored in heap, heap pointer is pushed.
Fieldload	heap	heap at field is pushed.
Fieldstore	stack	stack is popped to heap at field.
Newarray	length, values	length is store in heap, elements are initialized to values, heap pointer is pushed.
Arrayload	heap	heap at array element is pushed.
Arraystore	stack	stack is popped to heap at array element.
If	stack	stack is popped, if comparison succeeded; or it is false, otherwise.
Ifcmp	stack	stack is popped twice, if comparison succeeded; or it is false otherwise.
Invoke	stack	stack is popped to local variables.
Return	stack	stack is popped to return variable.
Pop	stack	stack is popped.
Dup	stack	stack is duplicated.
Swap	stack	two top elements are swapped.
Monitorenter	heap	monitor is entered, if succeeded; or it is false, otherwise.
Monitorexit	heap	monitor is exited.

Table 5.3: The summary of experiments: jMoped BDD library.

Types	Time (s)	Heap sizes
Push	8	170
Load	25	227
Store	40	237
Globalload	27	252
Globalstore	35	262
Unary	388	308
Binary	3517	433
Inc	146	224
New	20984	812
Fieldload	10067	670
Fieldstore	10426	800
Newarray	36531	831
Arrayload	161755	1331
Arraystore	101681	1382
If	6	161
Ifcmp	151	259
Invoke	36	235
Return	82	260
Pop	7	142
Dup	529	373
Swap	10302	575
Monitorenter	8787	734
Monitorexit	5396	719

Table 5.4: Experimental results: jMoped BDD library.

Sizes of `x` (bits)			1	2	3	4	5	6	7	8
Java 5.0	Eager	T (s)	9.3	10.8	16.9	31.1	67.9	117.8	225.7	457.9
		N (10^6)	0.4	0.5	0.8	1.4	2.5	5.2	9.0	18.1
		VT	48	87	167	327	648	1348	2567	5126
	Lazy	T (s)	19.7	17.7	19.6	17.5	17.2	18.9	16.7	18.8
		N (10^6)	1.2	1.2	1.2	1.3	1.2	1.3	1.3	1.3
		VT	3	3	3	3	3	3	3	3
Java 6.0	Eager	T (s)	15.1	18.6	37.5	64.3	147.7	301.7	642.0	1732.0
		N (10^6)	0.4	0.7	1.1	2.0	3.7	7.1	13.9	27.9
		VT	105	209	417	833	1655	3329	6657	13313
	Lazy	T (s)	20.9	20.8	19.4	22.3	20.8	18.8	23.4	23.2
		N (10^6)	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
		VT	3	3	3	3	3	3	3	3

Table 5.5: Experimental results: `java.util.Vector` class.

However, in Java 5.0, the constructor of `v2` is not atomic, and as a result the assertion can be violated, i.e. `v2` is not empty but `x` is not inside `v2`. jMoped detects this bug. The first half of Table 5.5 shows the time until the bug is found (T), the number of BDD nodes required (N), and the number of view tuples inspected (VT) in several experiments. In all experiments the bit size of all variables except `x` is set to 8, the heap size to 50 blocks, and the context bound to 3. The experiments differ on the size of `x` (1 to 8 bits), and on the splitting mode (eager or lazy).

In the current version of Java (version 6.0), the bug has been fixed. We reran all experiments with Java 6.0 and verified that, within the given bounds, the assertion is not violated. The second half of Table 5.5 presents the results.

Notice that the behavior of the program is independent of the value of `x`. The lazy approach benefits from this fact, and does not split at all when switching contexts. Therefore, the running time remains essentially constant when the number of bits of `x` increases. On the other hand, the time for eager splitting increases exponentially. However, the eager approach is faster and requires fewer BDD nodes when `x` is small. One of the reasons is that the lazy approach requires an extra copy of globals for keeping relations between current values of globals and initial values when the thread is awakened, which results in bigger BDDs.

One could argue that, since the Java 5.0 bug is already detected when `x` has 1 bit, the lazy approach does not give any advantage in this case. For Java 6.0, however, the analysis of larger ranges provides more confidence in the correctness of the code, and here the lazy approach clearly outperforms eager splitting.

Finally, we remark that the experiment is not as small as it seems. While the test method has only a few lines of code, the class `Vector` actually involves around 130 classes which together translate into a pushdown network of 30,000 rules. We are able to automatically translate all classes without any manipulations except in `java.lang.System`, where the method `arraycopy` is implemented in native code. The method `arraycopy` takes two arrays as parameters, and copies contents of an array to the other array. We need to manually create a stub in this case.

5.5 Windows NT Bluetooth driver

In this experiment, we consider three versions of a Windows NT Bluetooth driver [48, 11]. Figure 5.6 shows a Java implementation of the second version. All three versions follow the same idea and differ only in some implementation details. They all use the following class `Device`, which contains four fields:

```
int pendingIo; boolean stopFlag, stopEvent, stopped;
Device() {
    pendingIo = 1; stopFlag = stopEvent = stopped = false;
}
```

- `pendingIo` counts the number of threads that are currently executing in the driver. It is initialized to one in the constructor, increased by one when a new thread enters the driver, and decreased by one when a thread leaves.
- `stopFlag` becomes true when a thread tries to stop the driver.
- `stopEvent` models a stopping event, fired when `pendingIo` becomes zero. The field is initialized to false and set to true when the event happens.
- `stopped` is introduced only to check a safety property. Initially false, it is set to true when the driver is successfully stopped.

The drivers has two types of threads, *stoppers* and *adders*. A stopper calls `stop` to halt the driver. It first sets `stopFlag` to true before decrementing `pendingIo` via a call to `dec`. The method `dec` fires the stopping event when `pendingIo` is zero. An adder calls the method `add` to perform I/O in the driver. It calls the method `inc` to increment `pendingIo`; `inc` returns a successful status if `stopFlag` is not yet set. It then asserts that `stopped` is false before start performing I/O in the driver. The adder decrements `pendingIo` before exiting.

```
static void add(Device d) {
    int status = inc(d);
    if (status > 0) {
        assert(!d.stopped);
        // Performs I/O
    }
    dec(d);
}
static void stop(Device d) {
    d.stopFlag = true;
    dec(d);
    while (!d.stopEvent) {}
    d.stopped = true;
}
static int inc(Device d) {
    int status;
    synchronized (d) {
        d.pendingIo++;
    }
    if (d.stopFlag) {
        dec(d);
        status = -1;
    } else status = 1;
    return status;
}
static void dec(Device d) {
    int pio;
    synchronized (d) {
        d.pendingIo--;
        pio = d.pendingIo;
    }
    if (pio == 0)
        d.stopEvent = true;
}
```

Figure 5.6: Version 2 of Bluetooth driver

In the first version of the driver, the method `inc` was implemented differently:

```
private int inc(Device d) {
    if (d.stopFlag) return -1;
    synchronized (d) { d.pendingIo++; }
    return 0;
}
```

Moreover, the if-statement in `add` reads `if (status == 0)`. [48] reports a race condition for this version, which occurs when the adder first runs until it checks the value of `stopFlag`. Then, the stopper thread runs until the end, where it successfully stops the driver. When the context switches back to the adder, it returns from `inc` with status zero and finds out that the assertion is violated.

In [11] a bug in the second version of the driver was reported. The bug only occurs in the presence of at least two adders, and four context switches are required to unveil it: (i) The first adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The stopper sets `stopFlag` to true, decreases `pendingIo` back to 1, and waits for the stopping event. (iii) The second adder increases `pendingIo` to 2. However, since `stopFlag` is already set it decreases `pendingIo` back to 1 again. It returns from `inc` with status -1, which makes `pendingIo` become 0 and fires the stopping event. (iv) The stopper acknowledges the stopping event and sets `stopped` to true. (v) The first adder violates the assertion. Note that the bug can also be found in a slightly different manner where the second adder starts before the stopper.

The third version moves `dec(d)` inside the if-block in the method `add`. This eliminates the bug for the case with two adders and one stopper. However, jMoped found another assertion violation for one adder and two stoppers. We believe that this has not been previously reported, although it is less subtle than the previous bugs, requiring three context switches: (i) The adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The first stopper decreases `pendingIo` to 1. (iii) The second stopper decreases `pendingIo` to 0 and sets `stopped` to true. (iv) The adder violates the assertion.

Table 5.6 reports experimental results on these three versions. Notice that the lazy approach always involves fewer view tuples. This becomes more obvious when the number of contexts grows. As a consequence, we argue that by splitting lazily we can palliate explosions in the context-bounded reachability problem.

5.6 Binary search tree

We briefly give an intuition on the scalability of our approach by considering a binary search tree implementation [37] that supports concurrent manip-

	Version 1		Version 2		Version 3	
	Eager	Lazy	Eager	Lazy	Eager	Lazy
Time (s)	1.1	1.3	51.7	36.0	11.9	6.0
Nodes ($\times 10^3$)	46	88	720	1851	195	518
View tuples	21	4	1460	154	234	16
Contexts	3		5		4	

Table 5.6: Experimental results: Bluetooth drivers

ulations on trees. Unlike the previous two experiments, this algorithm is recursive.

Figure 5.7 presents the method `find` that searches for a node with value `v`, starting from node `n`, with `n.value` $\neq$ `v`. (The implementation makes sure that the value of the root is always greater than values of other nodes.) The method returns a node `f`, where `f.dir()` points to a node with value `v` if it exists, i.e. `f.dir().value` = `v`; otherwise `f.dir()` = `null`. Note that `f.dir()` is specified by `f.setDir(LEFT)` and `f.setDir(RIGHT)`, which set `f.dir()` to the left and right child of `f`, respectively. The node `f` is locked at the time when the method returns.

Note that after the node `f` is locked the method ensures that `f` is still the parent of `s`, i.e. it checks whether `s = f.dir()`. The check is necessary, since another thread might change `f.dir()` after deciding that `s = null` or `s.value = v`, and just before the statement `f.lock()`. In this case, the search must be resumed at node `f` again.

Figure 5.8 lists methods `search` and `insert`. Given a binary search tree, the method `search` searches for a node with value `v` in the tree. It simply calls the method `find`, and unlocks the returned node afterward. The method `insert` inserts a node with value `v` into the tree. It calls `find` to traverse the tree and find the right position of the new node. The method does nothing if a node with value `v` already exists in the tree. Otherwise, it creates a new node, and inserts it into the tree. The method `f.insert(w)` sets `f.dir()` to the node `w`.

We perform experiments with two types of threads, *inserter* and *searcher*. An inserter calls `insert` with a nondeterministic value, while a searcher calls `search` to search for the same value. We run jMoped with different numbers of inserters and searchers, and generate all reachable configurations within given contexts.

```
/**
 * Finds a node with value v in the tree, starting from node n.
 * If the node exists, the method returns a node f, where
 * f.dir() points to the node; otherwise f.dir() is null.
 * The node f is locked when the method returns.
 * @param n the node where the search starts.
 * @param v the searching value.
 * @return the parent node of the node with value v.
 */
private Node find(Node n, int v) {
    Node f = n;
    if (v < f.value)  f.setDir(LEFT);
    else f.setDir(RIGHT);
    Node s = f.dir();
    if (s != null && s.value != v)
        return find(s,v);
    else {
        f.lock();
        if (s != f.dir()) {
            // It slipped away, find again.
            f.unlock();
            return find(f,v);
        }
        return f;
    }
}
```

Figure 5.7: Binary search tree: the `find` method

```
/**
 * Searches for a node with value v in the binary search tree.
 * @param v the searching value.
 * @return the node with value v, if exist; or null, otherwise.
 */
public Node search(int v) {
    Node f = find(root, v);
    Node s = f.dir();
    f.unlock();
    return s;
}

/**
 * Insert a node with value v into the binary search tree.
 * The tree is unchanged if a node with value v already exists.
 * @param v the value of the new node.
 */
public void insert(int v) {
    Node f = find(root, v);
    if (f.dir() != null)
        f.unlock();
    else {
        Node w = new Node(v);
        f.insert(w);
        f.unlock();
    }
}
```

Figure 5.8: Binary search tree: the `search` and `insert` methods

Table 5.7 gives the running times. The numbers of threads are in the form $x+y$, where x and y are the numbers of inserters and searchers, respectively. The analysis took more than three hours in the case of $2+2$ threads with bound 6.

Threads	Contexts	Time (s)
$1+1$	3	3.8
$1+1$	4	8.3
$2+1$	4	127.1
$2+1$	5	712.3
$2+1$	6	5528.2
$2+2$	5	6488.0
$2+2$	6	timeout

Table 5.7: Experimental results: binary search trees

Chapter 6

Applications to SPKI/SDSI

SDSI, proposed by Lampson and Rivest [38], is a simple distributed security infrastructure that combines a simple public-key infrastructure design (SPKI) with a means of defining local name spaces. The combined SPKI/SDSI allows a principal to locally create groups of principals and delegate rights to other principals or groups of principals (without knowing individuals in the groups). In this chapter we introduce authorization and reputation systems based on SPKI/SDSI, and connect them to pushdown models. Later, we show that problems of determining whether a principal is authorized to access a resource or a problem of determining how much trust a principal places in another principal boil down to reachability problems of pushdown models with different weights.

6.1 Authorization systems

In access control of shared resources, authorization systems allow to specify a security policy that assigns permissions to principals in the system. The *authorization problem* is, given a security policy, should a principal be allowed access to a specific resource? In frameworks such as SPKI/SDSI [20] and RT_0 [40], the security policy is expressed as a set of certificates. Principals are public keys. A certificate is an electronic document signed by a principal. Given a set of certificates, the authorization problem reduces to discovering a chain of certificates proving that a given principal is allowed to access a given resource. Jha and Reps [32] showed that a set of SPKI/SDSI certificates can be seen as a pushdown system, and that certificate-chain discovery reduces to pushdown reachability. The SPKI/SDSI specification also provides for so-called *threshold certificates*, allowing specifications whereby a principal can be granted access to a resource if he/she can produce authorizations from multiple sources. We observe that this extension reduces to reachability on *alternating* pushdown systems.

We proceed in two steps. First, we present a subset of SPKI/SDSI that has been considered in most of the work on this topic. This subset of SPKI/SDSI does not handle threshold certificates, which we present in the second part.

6.1.1 SPKI/SDSI

In this thesis, we introduce only the basic notations that are required to understand SPKI/SDSI and its connections with alternating pushdown systems. A more thorough explanation can be found in [32].

SPKI/SDSI was designed to express authorization policies in a distributed environment. A central notion of SPKI/SDSI are *principals*. A principal can be a person or an organization. Each principal defines his/her own namespace, which assigns *rôles* to (other) principals. For instance, principal *Fred* can define the rôle `friend` and associate principal *George* with this rôle. Such associations are made in SPKI/SDSI by issuing so-called *name certificates* (*name certs*, for short). A special feature is that principals may reference the namespace of other principals in their certificates. For instance, *Fred* may state that all of *George*'s friends are also his own friends. In this way, SPKI/SDSI allows to associate a rôle with a group of principals described in a symbolic and distributed manner. SPKI/SDSI then allows to assign permissions to rôles using so-called *authorization certificates* (or *auth certs*).

The SPKI/SDSI standard also uses a public-key infrastructure that allows for certificates to be signed and verified for authenticity. Public-key infrastructure does not play a major rôle in our approach, but we shall re-use the ideas behind its naming scheme.

More formally, a SPKI/SDSI system can be seen as a tuple (P, A, C), where P is a set of *principals*, A is a set of *rôle identifiers* (or identifiers, for short) and $C = Na \uplus Au$ is a set of certificates. Certificates can be either *name certificates* (contained in Na), or *authorization certificates* (contained in Au).

A *term* is formed by a principal followed by zero or more identifiers, i.e., an element of the set PA^*. A term t is interpreted as denoting a set of principals, written $[\![t]\!]$, which are defined by the set of name certificates (see below).

A name certificate is of the form $p\ \mathtt{a} \rightarrow t$, where p is a principal, $\mathtt{a}$ is an identifier, and t is a term. Notice that $p\ \mathtt{a}$ itself is a term. The sets $[\![t]\!]$, for all terms t, are the smallest sets satisfying the following constraints:

- if $t = p$ for some principal p, then $[\![t]\!] = \{p\}$;
- if $t = t'\ \mathtt{a}$, then for all $p \in [\![t']\!]$ we have $[\![p\ \mathtt{a}]\!] \subseteq [\![t]\!]$;
- if $p\ \mathtt{a} \rightarrow t$ is a name certificate, then $[\![t]\!] \subseteq [\![p\ \mathtt{a}]\!]$.

For instance, Let *Fred*, *George*, and *Henry* be principals and `friend` be an identifier. Consider the following certificates.

$$\begin{aligned} \mathit{Fred}\ \mathtt{friend} &\rightarrow \mathit{George} && (6.1)\\ \mathit{Henry}\ \mathtt{friend} &\rightarrow \mathit{Fred} && (6.2)\\ \mathit{George}\ \mathtt{friend} &\rightarrow \mathit{Henry}\ \mathtt{friend} && (6.3)\\ \mathit{Henry}\ \mathtt{friend} &\rightarrow \mathit{Henry}\ \mathtt{friend}\ \mathtt{friend} && (6.4) \end{aligned}$$

The certificates (6.1) and (6.2) express that George is a friend of Fred and Fred is a friend of Henry, respectively, and (6.3) means that all of Henry's friends are also George's friends, and (6.4) says that the friends of Henry's friends are also his friends.

An authorization certificate has the form $p\,\square \rightarrow t\,b$, where p is a principal, t is a term, and b is either $\square$ or $\blacksquare$. Such a certificate denotes that p grants some authorization to all principals in $[\![t]\!]$. If $b = \square$, then the principals in $[\![t]\!]$ are allowed to delegate said authorization to others; if $b = \blacksquare$, then they are not. (Auth certs in SPKI/SDSI contain more details about the authorization that they confer; this detail is not important for our approach.)

Formally, auth certs define a smallest relation $\mathit{aut} \subseteq P \times P$ between principals such that $\mathit{aut}(p, p')$ holds iff p grants an authorization to p':

- if there is an auth cert $p\ \square \rightarrow t\ b$, for $b \in \{\square, \blacksquare\}$, and $p' \in [\![t]\!]$, then $\mathit{aut}(p, p')$;
- if there is an auth cert $p\ \square \rightarrow t\ \square$, $p' \in [\![t]\!]$, and $\mathit{aut}(p', p'')$, then $\mathit{aut}(p, p'')$.

For instance, the certificate

$$\textit{Fred}\ \square \rightarrow \textit{George}\ \texttt{friend}\ \blacksquare \tag{6.5}$$

means that Fred grants some right to all of George's friends, however, they friends are not allowed to delegate that right to other principals.

The *authorization problem* in SPKI/SDSI is to determine, given a system (P, A, C) and two principals p and p', whether p' is granted authorization by p, i.e., whether $aut(p, p')$.

SPKI/SDSI and pushdown systems

Certificates in SPKI/SDSI can be interpreted as prefix rewrite systems. For instance, if $p\ \texttt{a} \rightarrow p'\ \texttt{b c}$ and $p'\ \texttt{b} \rightarrow p''\ \texttt{d e}$ are two certificates interpreted as rewrite rules, then their concatenation rewrites $p\ \texttt{a}$ to $p''\ \texttt{d e c}$. In SPKI/SDSI, a concatenation of two or more certificates is called a *certificate chain*. It is easy to see that the authorization problem, given principals p and p', reduces to the problem of whether there exists a certificate chain that rewrites $p\ \square$ into either $p'\ \square$ or $p'\ \blacksquare$ (in the first case, p' also has the right to delegate the authorization further, in the second case he has not). Consider as an example the certificates (6.1)–(6.5) above. It can be shown that Fred grants a right to George by the following certificate chain.

$$\begin{aligned}
\textit{Fred}\ \square \ &\xrightarrow{(6.5)}\ \textit{George}\ \texttt{friend}\ \blacksquare\\
&\xrightarrow{(6.3)}\ \textit{Henry}\ \texttt{friend}\ \blacksquare\\
&\xrightarrow{(6.4)}\ \textit{Henry}\ \texttt{friend}\ \texttt{friend}\ \blacksquare\\
&\xrightarrow{(6.2)}\ \textit{Fred}\ \texttt{friend}\ \blacksquare\\
&\xrightarrow{(6.1)}\ \textit{George}\ \blacksquare
\end{aligned}$$

Moreover, it is well-known that the type of rewrite systems induced by a set of SPKI/SDSI certificates is equivalent to that of a pushdown system, see, e.g. [32, 54, 57, 33]. For example, a cert like $p\ \texttt{a} \rightarrow p'\ \texttt{b c}$ is interpreted as a pushdown transition, where p, p' are states of the finite control and where the stack content $\texttt{a}$ is replaced by $\texttt{bc}$. More precisely, a system (P, A, C) corresponds to a pushdown system $(P, A \cup \{\blacksquare, \square\}, C)$, where P is re-interpreted as the set of *control locations*, $A \cup \{\blacksquare, \square\}$ as the *stack alphabet*, and C as the set of *transitions* of the pushdown system. Then, the SPKI/SDSI authorization problem reduces to a pushdown reachability problem, i.e., whether from control location p with the symbol $\square$ on the stack (and nothing else) one can eventually reach control location p' with either $\blacksquare$ or $\square$ on the stack.

6.1.2 Intersection certificates

The SPKI/SDSI standard provides for so-called *threshold certificates*, which consist of, say, given $n > 1$ an auth cert of the form

$$p\ \square \rightarrow \{t_1 b_1, \ldots, t_n b_n\}, \text{ where } b_1, \ldots, b_n \in \{\square, \blacksquare\}\ , \tag{6.6}$$

and an integer $k \leq n$. The meaning of such a cert is that p grants authorization to principal p' if there is a certificate chain to p' from at least k out of $t_1 b_1, \ldots, t_n b_n$. We restrict ourselves to the case where $k = n$ and use the more suggestive name *intersection certificate* instead. Notice that threshold certificates for name certs could be defined analogously. However, we restrict the use of threshold certificates to just auth certs, following the claim from [15, 32] that the use of threshold certificates in name certs would make the semantics "almost surely too convoluted".

If intersection certificates are involved, proofs of authorization can no longer be done purely by certificate chains. Instead, a proof becomes a *certificate tree*, where the nodes are labeled with terms and the edges with rewrite rules that can be applied to the term labeling their source nodes. The root is of the form $p\ \square$, and if an intersection certificate is used to rewrite a node u, then the children of u are the elements of the right-hand side of the certificate. The tree is considered a valid proof of authorization for principal p' if all the children can be rewritten to $p'\ b$, where $b \in \{\square, \blacksquare\}$.

It can now easily be seen that in the presence of intersection certificates, the certificate set can be interpreted as an *alternating* pushdown system, and that the authorization problem reduces to the reachability analysis of the alternating pushdown system. In other words, p' is granted access to resource of p if it can be proved that $\langle p, \square\rangle \in pre^*(\{\langle p', \square\rangle, \langle p', \blacksquare\rangle\})$. Moreover, the certificate set always conforms the special case in Section 3.1.2, i.e. the resulting alternating pushdown system is *simple*, having $\{\square, \blacksquare\}$ as the set of bottom stack symbols. The reason can be readily seen, since only auth certs can result in more than one child node in the certificate tree, and they always of the form depicted in (6.6). Consequently, the pair of the alternating pushdown system and the set $\{\langle p', \square\rangle, \langle p', \blacksquare\rangle\}$ forms a *good instance*.

To apply the algorithm in Section 3.1.2, we assume without loss of generality that terms in name certs, non-intersection auth certs, and intersection auth certs contain at most two, one, and zero identifiers, respectively, and that the number of terms in intersection certs is two. Therefore, we have $C = C_0 \uplus C_1 \uplus C_2 \uplus C_t$, where C_t contains intersection certs, C_0 contains the name certs in which terms have zero identifiers, C_1 contains the name and non-intersection auth certs in which terms have one and zero identifiers, respectively, and C_2 consists of the rest. Let n be the number of different terms in C_0. The following theorem follows immediately from Lemma 3.4.

Theorem 6.1 *The authorization problem in a SPKI/SDSI system* (P, A, C) *can be solved in* $O(|C_0| + (|C_1| + |C_t|)n + |C_2|n^2)$ *time.*

6.1.3 Example and experiments

We have implemented a prototype of Algorithm 3.3 (in fact, a dedicated version for good instances) inside the Nexus platform [27]. Nexus is a platform that provides an infrastructure to support spatial-aware applications. An application can use Nexus "middleware" in order to obtain context data about mobile objects registered at the platform, like the position of an object or whether it enjoys a given relation to another object.

Nexus is based on an *Augmented World Model* (AWM). AWM can contain both real world objects (e.g. rooms or streets) and virtual objects (e.g. websites). Furthermore, Nexus defines a language called *Augmented World Modeling Language* (AWML). This XML-based language is used for exchanging Nexus objects between the platform and data repositories.

Our prototype extends the AWM and AWML with name and authorization relations, which can be viewed as name and authorization certificates in the case of SPKI/SDSI, respectively. In other words, we model relations as virtual objects in the Nexus context. Moreover, we extend the platform so that it can serve applications querying relations between entities. Note that, normally, the base information about objects is contained in a Nexus database (the so-called context server) and returned in the form of AWML documents. Our prototype is not yet connected to such a database; instead, all data is kept directly in AWML.

A scenario

Consider a scenario where company X takes part in a trade fair. The exhibition center consists of 2 exhibitions. An exhibition's area is a hierarchical structure with 3 exhibition halls, divided into 4 floors with 5 booths each. The structure can be written by pushdown rules as follows, given that $1 \leq i \leq 2, 1 \leq j \leq 3, 1 \leq k \leq 4, 1 \leq l \leq 5$:

$$E_i \texttt{ Area} \rightarrow E_i \texttt{ Hall Floor Booth} \tag{6.7}$$
$$E_i \texttt{ Hall} \rightarrow H_{[i,j]} \tag{6.8}$$
$$H_{[i,j]} \texttt{ Floor} \rightarrow F_{[i,j,k]} \tag{6.9}$$
$$F_{[i,j,k]} \texttt{ Booth} \rightarrow B_{[i,j,k,l]} \tag{6.10}$$

Now, company X launches a promotion for visitors of the exhibition center to freely download ringtones for their mobile phones. The following visitors are allowed to download: (1) customers of X who are currently in the area of exhibition 1; (2) non-customers to whom the right has been delegated by one of X's customers; (3) customers who are currently not in the area of exhibition 1, but have received delegation from another visitor of exhibition 1. The company can express this authorization policy by the following rule:

$$K_X \ \square \rightarrow \{E_1 \texttt{ Area Visitor } \square, K_X \texttt{ Customer } \square\} \tag{6.11}$$

We consider varying numbers of visitors and customers in the following experiments. For instance, the facts that Alice is visiting a booth in exhibition 1, and that she delegates her right to Bob, who is a customer of X, can be written as:

$$B_{[1,j,k,l]} \texttt{ Visitor} \rightarrow K_{Alice}, \qquad \text{for some } j, k, l \tag{6.12}$$
$$K_{Alice} \ \square \rightarrow K_{Bob} \ \blacksquare \tag{6.13}$$
$$K_X \texttt{ Customer} \rightarrow K_{Bob} \tag{6.14}$$

When Bob wants to download a ringtone, we can efficiently compute the set $pre^*(\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\})$ by noting the fact that the rules (6.7)–(6.14) and $\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\}$ form a good instance. Bob's request is granted in this case because $\langle X, \square\rangle \in pre^*(\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\})$. Note that Bob can only download as long as Alice stays in booths in the exhibition 1. As soon as she moves away (i.e. the rule (6.12) is removed), a request from Bob can no longer be granted even though he is a customer of X.

The above scenario is implemented as an application of the Nexus platform. We report on the running time for some experiments. The experiments should give a rough idea of the size of problems that can be handled in reasonable time.

We randomly add visitors to the exhibition center, and let them randomly issue certificates. Requests for ringtones are also made randomly. For each request, we measure time needed for finding an evidential certificate chain, i.e. the time needed for deciding whether the request should be granted or not. We consider a base case with 1000 visitors in the exhibition center, 100 of them are customers of the company X, and the visitors issue 1000 authorization certificates. The issuer of a certificate decides randomly whether the right can be further delegated or not. The series were conducted on a 2 GHz PC with 256 MB RAM.

Experiment 1

In the base case, 10 % of visitors are customers of X, and a visitor issues one certificate on average. In our first experiment we keep these two ratios constant, and increase the number of visitors (for example, if there are 2000 visitors, there will be 200 customers that authorize 2000 times). We ran the experiment five times for each set of parameters. In each run 1000 random download requests are made. Table 6.1 displays the average results for 1000, 2000, 5000, and 10000 visitors (V). The table shows how often the request was granted (G) and rejected (R), the average time of a certificate search (T), and average time for granted (T(G)) and rejected (T(R)) searches. All measurements are in milliseconds.

In a realistic scenario, solving the authorization problem requires to query databases (e.g. databases containing the positions of objects) and transmit data over a network, which are comparatively expensive operations. We kept relations of various types in different AWML files and whenever a piece of data was needed, we retrieved it from there. Since opening and reading files is also a comparatively expensive operation, this gives some insight as to the overhead such operations would incur in practice. The table shows the number of times AWML files (F) needed to be opened in average. For comparison, the numbers for granted (F(G)) and rejected (F(R)) requests are also displayed.

This experiment allows to draw a first conclusion: The average time of a search does not depend on the number of visitors per se. When a visitor requests a download, the algorithm has to search for the issuers of its certificates. Since the number of certificates is equal to the number of visitors, each visitor has one certificate in average.

V	G	R	T	T(G)	T(R)	F	F(G)	F(R)
1000	229.8	770.2	18.71	29.09	15.49	13.84	22.54	11.19
2000	195.6	804.4	19.23	28.76	16.92	13.14	21.25	11.16
5000	202.2	797.8	18.62	29.33	15.90	12.99	21.10	10.93
10000	199.4	800.6	24.90	38.25	21.60	13.00	22.00	10.77

Table 6.1: Results of Experiment 1, time (T) measured in milliseconds

C	G	R	T	T(G)	T(R)	F	F(G)	F(R)
1000	23.0	77.0	18.71	29.09	15.49	13.84	22.54	11.19
2000	56.2	43.8	120.72	193.93	21.96	74.68	118.50	15.83
3000	86.4	13.6	1477.35	1704.21	33.66	625.41	721.69	12.91
4000	95.2	4.8	2279.13	2393.81	13.40	898.01	942.94	9.64

Table 6.2: Results of Experiment 2, time (T) measured in milliseconds

Experiment 2

In this experiment, we kept the number of visitors constant, and increased the number of certificates they issue, shown in column C in Table 6.2. The other columns are as in Experiment 1. Again, we ran the experiment five times for each value of C. Each run consisted of 100 random requests.

We see that the running time grows rapidly with the number of certificates issued. The explanation is the larger number of certificates received by each visitor, which leads to many more certificate chains. Observe also that the number of granted requests increases.

The overall conclusion of the two experiments is that the algorithm scales well to realistic numbers of visitors and certificates. Notice that in the intended application a user will be willing to wait for a few seconds.

6.2 Reputation system

A reputation system is a system that holds opinions or trusts that participants in a community have in each other, and attempts to determine a rating for each participant. A reputation system can be simple such as in eBay, where a pair of users can rate each other after they complete a transaction. A user's reputation consists of the number of positive and negative ratings that the user obtained in the history. The most successful reputation system is perhaps Google's PageRank [45]. In PageRank, web pages are ranked based on numbers of incoming links they have and the rankings of web pages where the links come from. In other words, a web page can "recommend" other web pages if it has links to them, and the "quality" of the recommendations depends on the reputation of the web page itself.

In the section, we present our reputation system, SDSIrep, which extends SPKI/SDSI with weights on certificates. SDSIrep can be used to build a reputation system suitable for modeling trust relationships in an open-world scenario. Moreover, SDSIrep allows to distinguish between the trust one has in a person and in their recommendations. We then show how these trust values can be aggregated to measure each participant's reputation.

We also expose the relationship between SDSIrep and probabilistic pushdown systems, and extend the probabilistic approach to *alternating* pushdown systems. This solution allows to handle intersection certificates in SDSIrep, therefore increasing the expressiveness of the framework at practically no extra computational cost. As a small case study, we propose a system for measuring academic reputation. We implement the algorithms for computing reputations in this example and report on their performance.

6.2.1 SDSIrep

We now introduce the model of trust and reputation employed by SDSIrep, which later motivates the design of our system. We then proceed to show how to compute trust and reputation values in this system.

A numerical model of trust

Many reputation systems allow participants to express degrees of trust numerically. A common problem with this is that malicious participants may attempt to "spam" the system and boost each other's reputations with arbitrarily high values. The solution employed here is to normalize trust values. In SDSIrep, each principal has a total trust of 1 at his/her disposal, fractions of which can be allocated freely to other principals.

Like in EigenTrust [36], this approach lends itself to a probabilistic interpretation, similar to the "Random Surfer" model used in PageRank. We interpret a SDSIrep system as a Markov chain whose states are the participants, and where the trust that participant A has in B (expressed as a fraction between 0 and 1) serves as the probability of going from A to B. Then, one way to find reputable participants is to perform a random walk on this Markov chain: after a “long enough” period of time, one is more likely to be at a well-reputed participant than not. In particular, each party’s reputation is taken as their value in the stationary vector of the Markov chain. Thus, even though all participants can distribute a total trust value of 1 to others, this does mean that the opinions of all participants have the same influence. Well-reputed participants will be visited more often in a random walk than less-reputed ones, giving more weight to their opinions.

What distinguishes SDSIrep from EigenTrust is the way peer-to-peer ratings are specified: principals can assign their trust to *groups of* principals that are defined indirectly, using name certificates like in SPKI/SDSI. Membership in a group is associated with a numeric value, in a kind of fuzzy logic. Suppose, for instance, that a researcher wants to recommend those researchers whose findings have been published in a certain journal. Then, somebody with 10 papers in that journal could be considered to belong more strongly to that group than somebody with just one paper. SDSIrep allows to make such distinctions.

In the terminology of [35], PageRank, EigenTrust, and SDSIrep are all examples of *flow models*. In a flow model, participants can only increase their reputation at the cost of others. This property is obviously satisfied by SDSIrep, because the sum of the reputation values over all participants is bounded by 1. Thus, the absolute reputation values computed within the SDSIrep framework have no meaning in themselves; they only indicate how well-reputed each participant is in comparison with others.

SDSIrep certificates

Our system is based on the design of SPKI/SDSI, i.e. a SDSIrep system is again a triple (P, A, C) with (almost) the same meaning as in Section 6.1.1. However, in SDSIrep, we are not concerned with authorization problems. Rather, we reinterpret authorization certificates as *recommendations*, which express trust in certain groups of principals.

Another change is the addition of weights to certificates. Adding weights drawn from the set $[0, 1]$ to recommendation certs allows to express the degree of recommendations. Similarly, weights on name certs express the degree of membership to a set. We provide only simple examples in this section; a more elaborate example of a SDSIrep system is presented in Section 6.2.3.

Weighted recommendation certs allow to recommend all members of a group by issuing one single cert. This reflects common situations in which a principal recommends a group even though the members of the group change along time, or even though he or she does not know many of its members.

A weighted recommendation cert has the form $p \,\square \xrightarrow{x} t \,\blacksquare$, where $x \in [0,1]$ is its weight. Such a cert states that the principal p recommends the principals of the set $[\![t]\!]$ with weight x. The cert $p \,\square \xrightarrow{x} t \,\square$ states that p recommends not the principals of $[\![t]\!]$ themselves, but *their recommendations* with weight x.

As an example, suppose that researcher A wants to give 50% of his "share" of recommendations to the authors of journal J. This could be stated by the cert $A \,\square \xrightarrow{0.5} J\ \texttt{aut}\ \blacksquare$. To explain the semantic difference between $\square$ and $\blacksquare$, imagine a reputation system for film directors with directors and critics as principals. Film critics will not be recommended for their directing skills, only for their recommendations. A similar distinction exists in PGP, which separates the trust that principals have in the authenticity of some person's public key from the trust they have in the ability of that person to correctly judge the authenticity of other people's keys.

Notice that there is no certificate with $\blacksquare$ on the left-hand side. Thus, a chain starting with a recommendation cert of the form $p \,\square \xrightarrow{x} t \,\blacksquare$ necessarily ends when t has been rewritten to an element of $[\![t]\!]$, whereas a chain starting with $p \,\square \xrightarrow{x} t \,\square$ allows to apply further recommendation certs at that point. This corresponds to the idea that $\square$ expresses a recommendation of somebody's recommendations, whereas $\blacksquare$ expresses a recommendation of that person as such.

To normalize the trust values in the system, and in order to enable a probabilistic interpretation as discussed in this section, we additionally demand that the weights on each principal's recommendation certs add up to 1.

Weighted name certs have the form $p\ \texttt{a} \xrightarrow{x} t$, where $x \in [0,1]$. Intuitively, such a cert states a fuzzy membership relation: the elements of $[\![t]\!]$ belong to the set $[\![p\ \texttt{a}]\!]$ with membership degree x.

As an example, consider a journal J and an identifier `aut` such that $[\![J\ \texttt{aut}]\!]$ are the authors that have published in J. Then, if the journal has published 100 papers and B has authored 10 of them, B might be considered to belong to $[\![J\ \texttt{aut}]\!]$ with degree 10%, expressed as $J\ \texttt{aut} \xrightarrow{0.1} B$. In order to uphold the probabilistic interpretation we demand that for all pairs $p\ \texttt{a}$, the sum of the weights on all name certs with $p\ \texttt{a}$ on the left-hand side is 1.

Certificate chains and Markov chains

Consider the certs $A \square \xrightarrow{0.5} J\ \texttt{aut} \blacksquare$ and $J\ \texttt{aut} \xrightarrow{0.1} B$. If A gives 50% of his recommendations to the authors of J, and B has authored 10% of the papers in J, then a natural interpretation is that 5% of A's recommendations go to B. Thus, the weight of the certificate chain formed from the two certs is obtained by multiplying their individual weights.

To find out how much trust A puts into B, we are interested in the certificate chains going from $A \square$ to $B \blacksquare$. In general, there could be more than one such chain. Thus, one needs to find all these chains in order to determine the degree of recommendation A gives to B. The following example shows that the number of such paths can in fact be infinite:

$$A \square \xrightarrow{1} A\ \texttt{friend} \blacksquare \tag{6.15}$$
$$B \square \xrightarrow{1} A \blacksquare \tag{6.16}$$
$$A\ \texttt{friend} \xrightarrow{x} B \tag{6.17}$$
$$B\ \texttt{friend} \xrightarrow{1} A \tag{6.18}$$
$$A\ \texttt{friend} \xrightarrow{1-x} A\ \texttt{friend}\ \texttt{friend} \tag{6.19}$$

Cert (6.19) is the crucial one. It states that the friends of A's friends also belong to A's friends, albeit with smaller weight. Notice that whenever this cert can be applied, it can be applied arbitrarily often. So A recommends B through many possible chains: for instance, we can apply the cert (6.15), then cert (6.19) $2n$ times, and then certs (6.17) and (6.18) alternatingly n times each.

We can now define the two algorithmic problems related to SDSIrep. The *trust problem* in SDSIrep is as follows: Given two principals p and p', compute the sum of the weights of all certificate chains that rewrite $p \square$ into $p' \blacksquare$. The *reputation problem* is to compute, for each principal, their value in the stationary vector of the Markov chain in which the transition probabilities are given by the solutions to the pairwise trust problems. Next, we discuss solutions for the trust and reputation problems.

Solving the trust and reputation problems

It is easy to see that a system of SDSIrep certificates corresponds to a weighted pushdown system with the semiring $([0,1], +, \times, 0, 1)$. This type of weighted pushdown system is usually called *probabilistic* pushdown system. The trust problem in SDSIrep then reduces to a reachability problem on the probabilistic pushdown system, i.e., given p and p', compute the probability of reaching control location p' with stack content $\blacksquare$ when starting from p and $\square$.

Recall from Section 3.2 that the solution to this is given by an equation system. One can observe in this case that the semiring is ω-continuous (see Section 2.1.1), which ensures the existence of the least solution. We assume without loss of generality that terms in name and auth certs contain at most two and one identifiers, respectively. Given a SDSIrep system (P, A, C), the variables are elements of the set $\{\, [p, \mathtt{a}, q] \mid p, q \in P,\ \mathtt{a} \in A \cup \{\square, \blacksquare\} \,\}$, where $[p, \mathtt{a}, q]$ denotes the probability of rewriting the term $p\ \mathtt{a}$ into q. To solve the trust problem, we also add an artificial certificate $p'\ \blacksquare \xrightarrow{1} \bar{p}'$, where $\bar{p}'$ is a fresh control location; since $p'\ \blacksquare$ does not appear on any other left-hand side, the solution of $[p, \square, \bar{p}']$ gives us the trust placed by p in p'. Each variable $[p, \mathtt{a}, q]$ has the following equation:

$$[p, \mathtt{a}, q] = \sum_{p\mathtt{a} \xrightarrow{x} p'\mathtt{bc}} x \cdot \sum_{r \in P} [p', \mathtt{b}, r] \cdot [r, \mathtt{c}, q] + \sum_{p\mathtt{a} \xrightarrow{x} p'\mathtt{b}} x \cdot [p', \mathtt{b}, q] + \sum_{p\mathtt{a} \xrightarrow{x} q} x \tag{6.20}$$

Intuitively, equation (6.20) sums up the probabilities for all the possible ways of reaching q from $p\ \mathtt{a}$. We just explain the first half of the expression; the other cases are simpler and analogous: if $p\ \mathtt{a}$ is replaced by $p'\ \mathtt{b}\ \mathtt{c}$ (with probability x), then one first needs to rewrite this term to $r\ \mathtt{c}$ for some $r \in P$, which happens with the probability computed by $[p', \mathtt{b}, r]$, and then $r\ \mathtt{c}$ needs to be rewritten into q, which is expressed by the variable $[r, \mathtt{c}, q]$.

For instance, consider the system consisting of rules (6.15) to (6.19). Some of the resulting equations are (abbreviating `f` for `friend`):

$$\begin{array}{rcl} [B, \mathtt{f}, A] & = & 1 \qquad\qquad\qquad\qquad [B, \square, B] \;=\; 1 \cdot [A, \blacksquare, B] \\ [A, \mathtt{f}, B] & = & x + (1-x) \cdot ([A, \mathtt{f}, A] \cdot [A, \mathtt{f}, B] + [A, \mathtt{f}, B] \cdot [B, \mathtt{f}, B]) \end{array}$$

This equation system has a least solution, and the elements of this least solution correspond to the aforementioned probabilities. Notice that the equation system is non-linear in general. We discuss the resulting algorithmic problems in more detail in Section 6.2.3. The following theorem now follows from the definitions and Theorem 3.6.

Theorem 6.2 *The solution to the trust problem for principals p and p' is equal to the value of variable $[p, \Box, \bar{p}']$ in the least solution of the equation system (6.20).*

In general, the least solution cannot be computed exactly, but can be approximated to an arbitrary degree of precision using standard fix-point computation methods. We give more details on this computation when discussing our experiments in Section 6.2.3. Notice that the equation system actually gives the probabilities (and hence the trust values) for all pairs of principals, therefore all values in the Markov chain used for solving the reputation problem can be obtained from just one fixpoint computation.

As discussed earlier, a measure of the "reputation" of principals in the system can be obtained by computing the stationary vector of the Markov chain whose states are the principals and whose transition probabilities are given by the solutions of the trust problems. Computing the stationary vector amounts to solving a linear equation system, using well-known techniques.

However, for the stationary vector to exist, the Markov chain needs to be irreducible and aperiodic. This is not guaranteed in general: e.g., if there is a clique of participants who trust only each other, the Markov chain contains a "sink", i.e., it is not irreducible. This type of problem is also encountered in other models based on random walks, e.g. EigenTrust or PageRank, and the solutions employed there also apply to SDSIrep. For instance, the irreducibility and aperiodicity constraint can be enforced by allowing the random walk to jump to random states at any move with small probability. Notice that the example in Section 6.2.3 does not exhibit this kind of problem; therefore, we did not use this trick in our experiments.

6.2.2 Intersection certificates

Sometimes one wishes to recommend principals belonging to the intersection of two or more groups. For instance, researcher A may wish to recommend those of his co-authors that have published in journal J. In SDSIrep, we model this by a certificate such as $A \Box \xrightarrow{x} \{A\ \texttt{coaut}\ \blacksquare,\ J\ \texttt{aut}\ \blacksquare\}$. In general, intersection certificates have the form $p \Box \xrightarrow{x} \{t_1\, b_1, \ldots t_n\, b_n\}$, where

$b_1, \ldots, b_n \in \{\square, \blacksquare\}$, and express that p recommends the set $\bigcap_{i=1}^{n} [\![t_i]\!]$ with weight x.

The trust problem for the case without intersection certs consists of computing the values of certificate chains. When intersection certs come into play, we need to think of certificate trees instead, where each node is labeled by a term, and a node labeled by term t has a set of children labeled by T if T is the result of applying a rewrite rule to t. For instance, if in addition to the previous intersection certificate we have $A\ \texttt{coaut} \xrightarrow{y} B$ and $J\ \texttt{aut} \xrightarrow{z} B$, then we have the following certificate tree:

$$A\ \square \xrightarrow{x} \begin{cases} A\ \texttt{coaut}\ \blacksquare & \xrightarrow{y} \quad B\ \blacksquare \\ \quad J\ \texttt{aut}\ \blacksquare & \xrightarrow{z} \quad B\ \blacksquare \end{cases}$$

In the probabilistic interpretation, the probability for this tree is $x \cdot y \cdot z$. Thus, the *trust problem for SDSIrep with intersection certificates* is as follows: Given principals p and p', compute the sum of the probabilities of all trees whose root is labeled by $p\ \square$ and all of whose children are labeled by $p'\ \blacksquare$. Notice that the solution for the associated *reputation problem* remains essentially unchanged, as the addition of intersection certs merely changes the way peer-to-peer trust is assigned.

Solving the trust problem with intersection certs

We now extend the equation system from Section 6.2.1 to the case of intersection certificates. By following a similar reasoning to Section 6.1.2, it is easy to see that a SDSIrep system with intersection certificates corresponds to a *simple* weighted alternating pushdown system with the semiring $([0,1], +, \times, \times, 0, 1, 1)$. Let $\Xi = \{\blacksquare, \square\}$. Since intersection is restricted to recommendation certs, the following important properties hold: (1) if $p\ \square$ is the root of a certificate tree, then all nodes are of the form $t\ b$, where $b \in \Xi$ and t does not contain any symbol from Ξ; (2) if a term t of a certificate tree has more than one child, $t = p\ \square$ for some p. It follows that if a term $p\ w$ is the root of a tree and w does not contain any occurrence of $\blacksquare$ or $\square$, then every term of the tree has at most one child, and so the tree has a unique leaf. We exploit this fact in our solution.

We assume without loss of generality that terms in name and auth certs contain at most two and one identifiers, respectively, and that the number of terms in intersection certs is two. Let (P, A, C) be a SDSIrep system with intersection certificates. The variables of the equation system are of the form $[p, \perp, q]$ or $[p, a, q]$, where $p, q \in P$, $\perp \in \Xi$, $a \in A$. The variable $[p, \perp, q]$ represents the probability of, starting at $p\ \perp$, eventually reaching a tree where all leaves are labeled with q. The variable $[p, a, q]$ represents the probability of, starting at $p\ a$, reaching a tree whose unique leaf (here we use the fact above) is labeled with q. We add (as in Section 6.2.1) an artificial rule $p' \blacksquare \xrightarrow{1} \overline{p}'$, which is the only rule consuming the $\blacksquare$ symbol.

With $([0,1], +, \times, \times, 0, 1, 1)$ as the extended semiring we have the following system of equations (Section 3.2.2), for $p, q \in P$ and $a \in A \cup \Xi$:

$$\begin{aligned}
[p, \mathtt{a}, q] \quad = \quad & \sum_{p\mathtt{a} \xrightarrow{x} p'\mathtt{bc}} x \cdot \sum_{r \in P} [p', \mathtt{b}, r] \cdot [r, \mathtt{c}, q] + \sum_{p\mathtt{a} \xrightarrow{x} p'\mathtt{b}} x \cdot [p', \mathtt{b}, q] + \sum_{p\mathtt{a} \xrightarrow{x} q} x \\
& + \sum_{pa \xrightarrow{x} \{p_1 a_1 \perp_1, p_2 a_2 \perp_2\}} x \cdot \sum_{q_1, q_2 \in P} \prod_{i=1}^{2} [p_i, a_i, q_i] \cdot [q_i, \perp_i, q] \qquad (6.21)
\end{aligned}$$

(Notice that if $a \in A$ then the last sum is equal to 0 by property (2) above.) Moreover, we set $[p, \varepsilon, q] = 1$ if $p = q$ and 0 otherwise.

Intuitively, equation (6.21) sums up the probabilities for all the possible ways of reaching q from $p\ a$. The idea of the first three sums of the expression is the same as in the case without alternation, see Section 6.2.1. The last sum handles alternating rules: if $p\ a$ is replaced by $\{p_1\ a_1 \perp_1, p_2\ a_2 \perp_2\}$ (with probability x), then one needs to rewrite for all $i \in [2]$, $p_i\ a_i$ to q_i for some $q_i \in P$, which happens with the probability computed by $[p_i, a_i, q_i]$, and then $q_i\ \perp_i$ needs to be rewritten into q, which is expressed by the variable $[q_i, \perp_i, q]$. The corresponding equation system also has the same properties as in Section 6.2.1 and can be solved in the same way. The following theorem directly follows from Theorem 3.8.

Theorem 6.3 *The solution to the trust problem for principals p and p' in a SDSIrep system with intersection certificates is equal to the solution of variable $[p, \square, \overline{p'}]$ in the least solution of the equation system (6.21).*

6.2.3 Example and experiments

For demonstration purposes, we have used SDSIrep to model a simple reputation system for the PC members of TACAS 2008. We have chosen this example because the reader is likely to be familiar with the sources of reputation in academia, in particular in computer science. We do not claim that our experiments say anything really relevant about the actual reputation of the PC members, in particular because part of the required data (the personal preferences of the PC members, see below) was not available to us.

In this section, we give some details on this system, and report on the performance of our solver for the equation systems given in Sections 6.2.1 and 6.2.2.

A small system for academic reputation

Principals and identifiers. The set of principals contains the 28 members of the TACAS programme committee, 6 of the main conferences on automated verification (CAV, ICALP, LICS, POPL, VMCAI, TACAS), and 3 rankings: the CiteSeer list of 10,000 top authors in computer science (year 2006) [13], denoted `CiteSeer`, the CiteSeer list of conferences and journals with the highest impact factors (year 2003) [14], denoted `Impact`, and the list of h-indices [26] for computer scientists (year 2007), denoted `H-index`. The identifiers are `aut`, `publ`, `coaut`, and `circ`, with the following fuzzy sets as intended meaning:

- $[\![c\ \mathtt{aut}]\!]$: researchers that publish in conference c;
- $[\![r\ \mathtt{publ}]\!]$: conferences in which researcher r has published;
- $[\![r\ \mathtt{coaut}]\!]$: r's co-authors;
- $[\![r\ \mathtt{circ}]\!]$: r's "circle", defined as r's coauthors, plus the coauthors of r's coauthors, and so on (the degree of membership to the circle will decrease with the "distance" to r).

Name certs. Some illustrative examples of the certs in our system are shown in Figure 6.1. For the sake of readability, we present them without having normalized the weights (normalized values are more difficult to read and compare). So, to set up the equation system, one has to take all the certs with the same tuple $p\ \mathtt{a}$ on the left-hand side, say $p\ \mathtt{a} \xrightarrow{x_1} t_1, \ldots, p\ a \xrightarrow{x_n} t_n$, and then replace each x_i by $x_i / \sum_{i=1}^{n} x_i$. In this way, all weights are normalized.

Two certs describe to which degree a PC member is an author of a conference and which share each conference has in the PC member's publication list. In both cases, the weight (before normalization) is the number of papers

$$\texttt{TACAS aut} \xrightarrow{10} \texttt{KL} \tag{6.22}$$
$$\texttt{KL publ} \xrightarrow{10} \texttt{TACAS} \tag{6.23}$$
$$\texttt{KL coaut} \xrightarrow{22} \texttt{PP} \tag{6.24}$$
$$\texttt{KL circ} \xrightarrow{0.8} \texttt{KL coaut} \tag{6.25}$$
$$\texttt{KL circ} \xrightarrow{0.2} \texttt{KL circ circ} \tag{6.26}$$
$$\texttt{Impact}\ \square \xrightarrow{1.24} \texttt{TACAS aut}\ \blacksquare \tag{6.27}$$
$$\texttt{H-index}\ \square \xrightarrow{34} \texttt{KL}\ \blacksquare \tag{6.28}$$
$$\texttt{CiteSeer}\ \square \xrightarrow{2023} \texttt{KL}\ \blacksquare \tag{6.29}$$
$$\texttt{KL}\ \square \xrightarrow{4} \texttt{KL publ aut}\ \blacksquare \tag{6.30}$$
$$\texttt{KL}\ \square \xrightarrow{3} \texttt{KL circ}\ \blacksquare \tag{6.31}$$
$$\texttt{KL}\ \square \xrightarrow{2} \texttt{Impact}\ \square \tag{6.32}$$
$$\texttt{KL}\ \square \xrightarrow{3} \{\texttt{CiteSeer}\ \square,\ \texttt{H-index}\ \square\} \tag{6.33}$$

Figure 6.1: Name and recommendation certificates for the example

the author has published in the conference, obtained from DBLP [39]. For instance, for TACAS and Kim Larsen (KL), we have certs (6.22) and (6.23).

Another set of certs describes which PC members are coauthors of each other. The weight is the number of jointly written papers, obtained again from DBLP. For instance, cert (6.24) denotes that KL has written 22 papers with PP.

Finally, each PC member has a circle of fellow researchers, composed of the member's coauthors, the coauthors of the member's coauthors, and so on. We define KL's circle by means of certs (6.25) and (6.26).

Recommendation certs. The system contains one recommendation cert for each conference, in which Impact recommends the authors of the conference with the weight given by its impact factor. For TACAS we have cert (6.27).

The next two certs, (6.28) and (6.29) express that the h-index and CiteSeer lists recommend a PC member (in this case KL) with a weight proportional to his h-index and to his number of citations in the list, respectively.

Finally, each PC member issues four more certs. The certs for `KL` are given in (6.30)–(6.33). Intuitively, they determine the weight with which `KL` wishes to recommend his circle, the authors of the conferences he publishes in, and how much trust he puts in the CiteSeer and h-index rankings. In a real system, each PC member would allocate the weights for his/her own certs; in our example we have assumed that all PC members give the same weights. In order to illustrate the use of intersection certs we have assumed that `KL` only recommends researchers on the basis of their ranking values if they appear in *both* CiteSeer's list and in the h-index list (6.33). Moreover, observe that in certs (6.32) and (6.33), `KL` places trust in the *recommendations* given by the rule targets (signified by $\Box$), whereas in the other rules he expresses trust in the principals themselves.

In the following two sections we describe the running times and some interesting aspects of solving the equation systems computing the reputation of each researcher. All experiments were performed on a Pentium 4 3.2 GHz machine with 3 GB memory.

Experiment 1

We have written a program which takes as input the set of SDSIrep certificates described above, generates the equation system of Section 6.2.2, and computes its solution. We can then compute the degree to which researchers recommend one another. From the result we build a Markov chain as described in Section 6.2.1. The stationary distribution of the Markov chain, given at the top of Table 6.3, can be interpreted as the (relative) reputation of each researcher when compared to the others in the system.

The lower part of Table 6.3 shows how the running time scales when the number of researchers is increased. For this experiment we have put together the PCs of TACAS, FOSSACS, and ESOP, with a total of 76 members, adding FOSSACS and ESOP to the list of conferences. We have computed the stationary distribution for subsets of 10, 20, . . . , 76 PC members. The first line of the table shows the number of variables in the system (which is also the number of equations), and the second shows the time required to solve it and compute the stationary distribution.

Notice that the equation system used here is non-linear. We solve it using Newton's iterative method, stopping when an iteration does not change any component of the solution by more than 0.0001.

`PB`	`EB`	`TB`	`RC`	`BC`	`BD`	`PG`	`OG`	`AG`	`FH`	`MH`	`JJ`	`KJ`	`JK`
26	18	19	78	45	6	56	60	30	19	45	19	5	23
`BK`	`MK`	`KL`	`NL`	`KN`	`PP`	`SR`	`CR`	`JR`	`AR`	`SS`	`SS`	`BS`	`LZ`
10	30	88	26	37	33	64	22	45	6	54	15	80	41

scientists	10	20	30	40	50	60	70	76
variables	627	1653	3089	4907	7126	9752	12777	14779
time (s)	0.47	2.07	6.85	12.55	23.90	44.89	78.35	106.55

Table 6.3: Stationary distribution for TACAS PC members (values multiplied by 1000) and statistics for different numbers of researchers.

Experiment 2

In contrast to other trust systems, in which trust is assigned from one individual to another, our choice of SDSIrep allows to assign trust measures to sets of principals using multiple levels of indirection. For instance, A can transfer trust to B even without knowing the fact that B is a coauthor of C, and C publishes in the same conference as A. This added expressiveness comes at a price. Certs with more than one identifier on the right-hand side, like (6.26) or (6.30), cause the resulting equation system to become non-linear. Likewise, intersection certs also give rise to non-linear equations.

On the other hand, if the system does not contain these two types of certs, the resulting equation system is linear, and instead of Newton's method more efficient techniques can be applied, e.g. the Gauß-Seidel method. See for instance [44] for a reference.

In the following, let us assume that intersection certs are absent. Consider cert (6.26). The certificate is "recursive" in the sense that it can be applied arbitrarily often in a certificate chain, rewriting `KL circ` to `KL circ`^{n}, for any $n \geq 1$. Thus, the length of terms to which `KL circ` can be rewritten is unbounded. (In pushdown terms, the "stack" can grow to an unbounded size.) If the set of certs is such that this effect cannot happen, then each term can be rewritten into only *finitely many* different other terms. Therefore, we can apply a process similar to that of "flattening" a pushdown system into a finite-state machine and derive a *larger*, but *linear*, equivalent equation system. If there are recursive certs, we can still choose an arbitrary bound on the length of terms and ignore the contributions of larger terms. In this case, the "unflattened" and "flattened" systems do not have the same solution, but the solution of the "flattened" system converges to the solution of the "unflattened" one when the bound increases.

This provokes the question of whether the performance of the equation solver can be improved by bounding the maximal term length, "flattening" the non-linear system into a linear one, and solving the linear system. In order to experimentally address this question, we again took the system described earlier in the section, but without cert (6.33). We fixed the maximal term depth to various numbers, computed the corresponding linear flattened systems, and solved them using the Gauß-Seidel method. (We omit the details, which are standard.)

	Unflattened	D 2	D 3	D 4	D 5	D 6	D 7	D 8
vars	2545	5320	7059	8798	10537	12276	14015	15754
time	5.83	1.23	3.32	6.39	10.34	18.78	32.18	42.97

Table 6.4: Size of equation system and running time for flattened systems

We found that in this example flattening works very well. Even with stack depth 2 we obtained a solution that differed from the one given by Newton's method by less than 1% and can be computed in 1.23 seconds instead of 5.83. Table 6.4 shows the results for stack depths up to 8 (D stands for depth), i.e. the size of the equation system obtained for each stack depth and the time required to solve it. Notice that in this case, the growth of the equation system as the stack depth grows is benign (only linear); in general, the growth could be exponential.

This result might suggest that using Newton's method could always be replaced by flattening in the absence of intersection certs. However, some caution is required. When we tried to repeat the experiment for the case with 76 researchers, our solver was able to solve the unflattened system within two minutes, but ran out of memory even for a flattened stack depth of 2.

6.3 Pushdown games

In [10] Cachat provided an algorithm for computing the winning positions of a player in a pushdown reachability game. It is straightforward to reformulate the algorithm in terms of *pre** computations for alternating pushdown systems. We do this, and apply the results of Section 3.1.2 to provide very precise upper bounds for the complexity of these problems.

A *pushdown game system* (PGS) is a tuple $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$, where $(P, \Gamma, \Delta_{\mathcal{G}})$ is a pushdown system and P_0, P_1 is a partition of P. A PGS defines a pushdown game graph $G_{\mathcal{G}} = (V, \rightarrow)$ where $V = P\Gamma^*$ is the set of all configurations, and $p\gamma v \rightarrow qwv$ for every $v \in \Gamma^*$ iff $(p, \gamma, q, w) \in \Delta_{\mathcal{G}}$. P_0 and P_1 induce a partition $V_0 = P_0\Gamma^*$ and $V_1 = P_1\Gamma^*$ on V. Intuitively, V_0 and V_1 are the nodes at which players 0 and 1 choose a move, respectively. Given a start configuration $\pi_0 \in V$, a play is a maximal (possibly infinite) path $\pi_0\pi_1\pi_2\ldots$ of $G_{\mathcal{G}}$; the transitions of the path are called *moves*; a move $\pi_i \rightarrow \pi_{i+1}$ is made by player 0 if $\pi_i \in V_0$; otherwise it is made by player 1.

The winning condition of a reachability game is a regular *goal set* of configurations $R \subseteq P\Gamma^*$. Player 0 wins those plays that visit some configuration of the goal set and also those that reach a deadlock for player 1. Player 1 wins the rest. We wish to compute the winning region for player 0, denoted by $Attr_0(R)$, i.e. the set of nodes from which player 0 can always force a visit to R or a deadlock for player 1. Formally [10]:

$$\begin{array}{rcl} Attr_0^0(R) & = & R \ , \\ Attr_0^{i+1}(R) & = & Attr_0^i(R) \cup \{u \in V_0 \mid \exists v : u \rightarrow v, v \in Attr_0^i(R)\} \\ & & \cup \{u \in V_1 \mid \forall v : u \rightarrow v \Rightarrow v \in Attr_0^i(R)\} \ , \\ Attr_0(R) & = & \bigcup_{i \in \mathbb{N}} Attr_0^i(R) \ . \end{array}$$

Given a PGS $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$, we define an alternating pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ as follows. For every $p \in P$ and $\gamma \in \Gamma$: if $p \in P_0$, then for every rule $\langle p, \gamma\rangle \hookrightarrow \langle q, w\rangle$ of $\Delta_{\mathcal{G}}$ add the rule $\langle p, \gamma\rangle \hookrightarrow \langle q, w\rangle$ to Δ; if $p \in P_1$ and S is the set of right-hand-side configurations of rules with $\langle p, \gamma\rangle$ as left-hand-side, then add $\langle p, \gamma\rangle \hookrightarrow S$ to Δ. It follows immediately from the definitions that $Attr_0(R) = pre^*{}_{\mathcal{P}}(R)$ (intuitively, if $c \in pre^*{}_{\mathcal{P}}(R)$ then $c \Rightarrow C$ for some $C \subseteq R$, and so player 0 can force the play into the set C). So we can use Algorithm 3.3 to compute $Attr_0(R)$. To derive the complexity bound, we apply Lemma 3.3:

Theorem 6.4 *Let $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$ be a PGS and a goal set R recognized by an alternating automaton $\mathcal{A}_R = (Q, \Gamma, \delta_0, P, F)$. An alternating automaton accepting the winning region can be computed in $O(|\delta_0| + |\Delta_0| + |\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|2^a)4^n)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$.*

In [10] an upper bound of $O(|\Delta| \cdot 2^{c \cdot |Q|^2})$ is given. Our algorithm runs in $O(|\Delta| \cdot 2^{c \cdot |Q|})$ time, and in fact Theorem 6.4 further reduces the exponent $c \cdot |Q|$ to $|P_\varepsilon| + |Q_{ni}|$. Typically, $|P_\varepsilon| + |Q_{ni}|$ is much smaller than $|Q|$. First, recall that, because of the definition of $\mathcal{P}$-automaton, we have $P \subseteq Q$. Moreover, goal sets often take the form $p_1\Gamma^* \cup \ldots \cup p_n\Gamma^*$, i.e., player 0 wins if the play hits one of the control states $p_1, \ldots, p_n$. In this case we can construct $\mathcal{A}_R$ with $|Q_{ni}| = 1$. Since $|P_\varepsilon|$ is typically much smaller than $|P|$, the parameter n is much smaller than $|Q|$.

Chapter 7

Conclusions

The main contribution of the thesis is the development of reachability analyses for three weighted pushdown models—pushdown systems, alternating pushdown systems, and pushdown networks—and their application to different areas. We have proposed different algorithms depending on types of weights the models are based upon. The algorithms together with a Java front-end have been optimized and implemented in jMoped, enabling Java testing on wider ranges of inputs than one could achieve in traditional testing. One can think of jMoped as a virtual machine that can symbolically execute bytecode instructions on all inputs in a single run. jMoped has been successfully applied to non-trivial programs, both sequential and multithreading. Notably, jMoped was successfully applied to a part of itself.

We believe that one of the reasons that software model checking has not been extensively used in practice is because of the steep learning curve of existing tools. jMoped, an Eclipse plug-in with a user-friendly interface, has been designed to require the minimum amount of effort from users. It allows users to easily test Java programs without knowing model-checking techniques behind it. Unlike many model checkers that return only yes or no after the end of an analysis, jMoped progressively shows results of the analysis almost immediately after the analysis starts. Users can stop the analysis at any time, and are still able to obtain partial results.

jMoped can be improved in many aspects. Basically, jMoped only works when bytecode instructions are available. This is, however, not always the case; especially when the Java library is involved. As a result, jMoped might not be able to test some programs, unless stubs are explicitly written. Besides, jMoped tends to be slow for programs with many objects, i.e., when large heaps are required. Heap in jMoped is straightforwardly represented as an array of fixed-size integers, whose elements are never garbage-collected. Interesting questions are perhaps, how to encode heaps in a more efficient way, or even how to approximate sizes of heap elements in order to reduce the heap sizes. We believe that more intelligent ways of heap management are necessary to boost applicability of the tool.

Additionally, we have applied reachability analyses on pushdown models to the area of authorization and reputation systems. Both systems can be modeled by alternating pushdown systems in a very similar manner, except that weights are of different types. The problem of determining whether a principal is authorized to access a given resource or the problem determining a reputation of a principal boil down to solving reachability problems. At first this fact seemed to prevent its application in real-life, since reachability analyses on alternating pushdown systems takes exponential time in general. We, however, have pointed out that since the alternating part of a model can only be resulted from intersection authorization (or recommendation) certificates, the resulting model always satisfies a special case in which analyses merely take polynomial time.

Still, experiments have shown that naive implementations of the algorithms can be inadequate for real-life applications, especially when a large number of principals is involved. The problem is more serious in reputation systems where systems of equations need to be solved. For instance, to solve a system of equations with 30000 variables (which is a reasonable figure, recall that the largest experiment in Section 6.2.3 involve around 15000 variables) with the Newton's method one would need to deal with 30000-by-30000 matrices. Assuming that each element in the matrices is stored as a 64-bit (or 8-byte) double precision floating point, a total of 7.2×10^9 bytes is required to store a single matrix. This amount of memory already exceeds available memory of today's standard computers, not to mention how to efficiently solve the equations if one were able to somehow store the matrices. We believe that tackling these problems is an interesting future work that would lead to a broader applicability of pushdown models in this area.

Bibliography

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. CAV*, 2001.

[2] H. R. Andersen. An introduction to binary decision diagrams. Technical report, Technical University of Denmark, 1997. Lecture Notes.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL*, pages 1–3, 2002.

[4] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. ICALP*, 2001.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: applications to software engineering. *Int. Journal on Software Tools for Technology Transfer*, 9:505–525, 2007.

[6] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. CONCUR*, 1997.

[7] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejček. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. FSTTCS*, LNCS 3821, pages 348–359, 2005.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[9] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. CONCUR*, LNCS 630, pages 123–137, 1992.

[10] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proc. ICALP*, LNCS 2380, pages 704–715, 2002.

[11] S. Chaki, E. M. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Proc. TACAS*, LNCS 3920, pages 334–349, 2006.

[12] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[13] CiteSeer. Top 10,000 cited authors in computer science, available at http://citeseer.ist.psu.edu/allcited.html.

[14] CiteSeer. Estimated impact of publication venues in computer science, available at http://citeseer.ist.psu.edu/impact.html.

[15] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9:2001, 2001.

[16] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. Springer-Verlag, 1982.

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

[18] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. ICSE*, pages 439–448. ACM Press, 2000.

[19] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. SPIN*, LNCS 1680, pages 261–276, 1999.

[20] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylönen. *RFC 2693: SPKI Certificate Theory*. The Internet Society, 1999.

[21] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. CAV*, LNCS 1855, pages 232–247, 2000.

[22] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proc. TACAS*, LNCS 3920, pages 489–503, 2006.

[23] J. Esparza, A. Kučera, and R. Mayr. Model checking probabilistic pushdown automata. In *Proc. LICS*. IEEE, 2004.

[24] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. CAV*, LNCS 2102, pages 324–336, 2001.

[25] K. Havelund. Java PathFinder, a translator from Java to Promela. In *Proc. SPIN*, LNCS 1680, page 152, 1999.

[26] J. E. Hirsch. An index to quantify an individual's scientific research output. *PNAS*, 102(46):16569–16572, 2005.

[27] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schwehm. Nexus - an open global infrastructure for spatial-aware applications. Technical Report 1999/02, Universität Stuttgart: SFB 627, 1999.

[28] G. J. Holzmann. Logic verification of ANSI-C code with Spin. In *Proc. SPIN*, LNCS 1885, pages 131–147, 2000.

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

[30] Java PathFinder, available at http://javapathfinder.sourceforge.net/.

[31] S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Proc. CSFW*, page 129. IEEE Computer Society, 2002.

[32] S. Jha and T. Reps. Model checking SPKI/SDSI. *JCS*, 12(3–4):317–353, 2004.

[33] S. Jha, S. Schwoon, H. Wang, and T. Reps. Weighted pushdown systems and trust-management systems. In *Proc. TACAS*, LNCS 3920, pages 1–26, 2006.

[34] jMoped. The tool's website, http://www7.in.tum.de/tools/jmoped/.

[35] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. In *Decision Support Systems*, 2005.

[36] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proc. WWW*, 2003.

[37] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[38] B. Lampson and R. L. Rivest. SDSI – a simple distributed security infrastructure, available at http://people.csail.mit.edu/rivest/sdsi11.html.

[39] M. Ley. DBLP bibliography, available at http://www.informatik.uni-trier.de/~ley/db/.

[40] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[41] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Prentice Hall, 2nd edition, 1999.

[42] K. L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[43] OpenJDK, available at http://community.java.net/openjdk/.

[44] J. M. Ortega. *Numerical Analysis: A Second Course*. Academic Press, 1972.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[46] ParForCE Project Workshop. Performance comparison between Prolog and Java, available at http://www.clip.dia.fi.upm.es/Projects/Par Force/Final_review/slides/intro/node4.html.

[47] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. TACAS*, LNCS 3440, pages 93–107, 2005.

[48] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Proc. PLDI*, pages 14–24. ACM, 2004.

[49] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Symposium on Programming*, LNCS 137, pages 337–351, 1982.

[50] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Programming Languages and Systems*, 22(2):416–430, 2000.

[51] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, 2005.

[52] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. FSE*, pages 267–276. ACM, 2003.

[53] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technische Universität München, 2002.

[54] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Proc. CSFW*, pages 202–218. IEEE, 2003.

[55] F. Somenzi. CUDD: CU decision diagram package release 2.4.1, available at http://vlsi.colorado.edu/~fabio/CUDD/.

[56] D. Suwimonteerabuth. Verifying Java bytecode with the Moped model checker. Master's thesis, Universität Stuttgart, 2004.

[57] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *Proc. ATVA*, LNCS 4218, pages 141–153, 2006.

[58] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[59] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[60] M. A. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley, 1999.

[61] J. Whaley. JavaBDD, available at http://javabdd.sourceforge.net/.