Masaryk University
Faculty of Informatics

Formal Verification of Sequential Systems with Infinitely Many States

Master's Thesis

Jan Obdržálek

2001

Declaration

I hereby declare that this thesis was composed by myself, and the work contained in it is my own, unless otherwise stated. I cite all the sources and literature used together with a reference to the corresponding source.

Acknowledgment

First of all, I would like to thank Antonín Kučera for his guidance and encouragement, and to Stefan Schwoon for providing me his LTL model checker. Special thanks go to my parents for their patience and constant support during my studies.

I also want to thank all members of the ParaDiSe laboratory and, last but not least, to Ate, a Greek goddess who makes men blind so that they will blunder into guilty acts.

Abstract

In recent years, model checking algorithms for verification of infinite-state systems were deeply studied and applied to practical problems. We show, how to use the algorithms for pushdown systems and various modal logics of [4] for verification of Java programs. The process of mechanical abstract model generation is described, and a prototype tool called JAVACHECK is implemented to verify our concept. We also present some examples of properties which can be verified using our approach.

Key words

Formal verification, Java, model checking, infinite-state systems, pushdown systems.

Contents

1 Introduction **1**
- 1.1 Semantic models 1
- 1.2 Approaches to formal verification 2
- 1.3 Verification of software 3
- 1.4 Contribution of the thesis 3

2 Basic Facts and Known Results **5**
- 2.1 Transition systems 5
- 2.2 Pushdown systems 6
- 2.3 Temporal logics 6
  - 2.3.1 Linear-time logics 7
  - 2.3.2 Branching-time logics 8
  - 2.3.3 Action-based vs state-based logics 10
- 2.4 Known results on the model checking problem 11
  - 2.4.1 Complexity results 11
  - 2.4.2 Available implementations 11

3 Modeling Java Using PDS **13**
- 3.1 Java overview 13
- 3.2 Programs with procedures as PDS 14
- 3.3 Intraprocedural analysis 16
- 3.4 Method call resolution 16
  - 3.4.1 Class hierarchy analysis 18
  - 3.4.2 Rapid type analysis 20
- 3.5 Exception handling 21
  - 3.5.1 Exceptions in Java 21
  - 3.5.2 Modeling the exception flow using PDS 22

4 The JAVACHECK Tool **26**
- 4.1 The JAVACHECK overview 27
  - 4.1.1 Data structures used 27
  - 4.1.2 JAVACHECK limitations 28
  - 4.1.3 Performance issues 28

4.2 Implementation details . . . 28
4.2.1 Method invocation resolution . . . 28
4.2.2 Class hierarchy analysis . . . 30
4.2.3 Finite automata support . . . 30
4.2.4 Interfacing with the model checker . . . 31
4.2.5 Control flow graph visualisation . . . 32
4.3 Usage and options . . . 32

5 Examples **34**
5.1 A graph plotting example . . . 34
5.2 A generic counter . . . 34
5.3 Application to Java security . . . 36
5.3.1 Electronic commerce example . . . 36

6 Conclusion, Possible Extensions and Related Work **40**
6.1 Possible extensions . . . 40
6.2 Related work . . . 41

Chapter 1

Introduction

The problem of *program verification* attracts an unceasing popularity nowadays. As software and hardware systems are becoming more and more complex, we are less certain about their correctness. But there are many applications, where we need to be absolutely sure about the program's behaviour (e.g. medical systems or air traffic control systems).

In order to be successful and regularly used, methods for program verification must be cheap, reliable, and easy to use. This can be achieved only by automating the process using computers. Computer aided program verification is therefore one of the most studied areas in computer science.

1.1 Semantic models

Various models of computation and its semantics were proposed, allowing us to reason about programs and their behaviour. Denotational semantics of Scott and Strachey [38] views every expression, statement and every program as a mathematical function transforming input parameters into output values. However, denotational semantics cannot express about *reactive systems*, whose correct behaviour is a nonterminating computation which maintains on-going interaction with its environment (operating systems, control programs for nuclear power plants or air traffic control systems all fall into this category).

The operational semantics in style of Plotkin [34] is much more usable in this context. The semantics of a program is given using the notion of *transition system*. Transition system is a graph structure, where nodes correspond to the states the system may enter and edges denote a single step of the computation.

1.2 Approaches to formal verification

There are many different approaches to the verification problem. The simplest, but the least effective, is *simulation* – programs are tested with respect to the limited set of input data. But as we cannot try out all possible combinations of input data, the simulation can easily miss serious design flaws.

A kind of "total verification" can be achieved using a technique called *equivalence checking*. Both the program specification and implementation are expressed in some mathematical model, and these two models are then mechanically checked to be equivalent. There are many behavioural equivalences used (see [41]), the most important being the *strong bisimulation* [33]. This technique is, unfortunately, limited by the following two factors: 1) the bisimulation equivalence problem is undecidable for many classes of transition systems, and 2) even if decidable, the complexity is usually very high or not even known.

Model checking (see [11] for more information on this topic) seems to be a powerful technique which can overcome some of these limitations. Instead of testing an implementation being behaviourally equivalent to a specification, we ask whether the system satisfies a given property. There was a need for some formalism able to state the properties the system must satisfy. As advocated by Pnueli [35], *temporal logics*, which can assert how the behaviour of a system evolves over time, are exactly the formalism we wanted.

The first temporal logic model checking algorithms were introduced in early 80's by Clarke and Emerson [10, 14]. This allowed the whole process to be automated. But there was a restriction on the size of systems being checked. Early algorithms were able to check transition graphs with between 10^4 and 10^6 states at a rate of about 100 states per second [11]. This was quite a big restriction, mainly because of the *state space explosion problem*. This problem is because of the fact that the number of states grows exponentially with the number of components run in parallel.

In 1987 McMillan observed [6] that using a symbolic representation of state-transition graphs, much larger systems can be verified. The symbolic representation was based on *ordered binary decision diagrams (OBDDs)* of Bryant [5]. This approach originally allowed graphs with more than 10^{20} states to be checked, and this number was increased to 10^{130} in the following years [9].

Model checking of *finite-state* systems proved to be considerably successful for verification of hardware systems. Since the half of 90's it is being regularly used by the hardware making industry to verify circuit designs. Typical examples of the tools being used are SPIN[23] and SMV[31].

1.3 Verification of software

Concerning software systems, the situation is very different. Firstly, there is a *model abstraction problem*: the semantic gap between formalisms used by software developers and those accepted by verification tools. Whereas programs are developed using general-purpose programming languages as C or Java, verification tools accept languages with simple semantics (e.g. process algebras).

Secondly, software systems are less structured then the hardware ones and the cardinality of data sets associated with variables makes the state space very big or even infinite. *Abstraction* techniques must be used to decrease the state space size to an acceptable level. Parts of the program code not related to the checked property are removed (this process is called *slicing*) and the domain of variables is restricted (e.g. an integer value is modeled by a boolean with zero/non-zero meaning).

Another problem is that finite-state models do not capture faithfully the procedural structure of a program. This is because of the presence of possibly infinite control and data structures, the most important being the call stack in programs with recursion. The use of method inlining to produce finite-state models of such programs necessarily prevents us from checking very important properties based on the call stack behaviour.

Infinite-state model checking promises to solve the latter problem. In the last few years, promising results for interesting classes of infinite-state systems and powerful logics appeared. Especially the class of pushdown systems was studied in detail – the results can be found in [4, 19, 7, 8, 43]. Recently, two model checkers for the class of pushdown systems were implemented. One for the LTL logic [15] and the second one for the alternation-free fragment of the modal μ-calculus [36].

1.4 Contribution of the thesis

We investigate the possibility of an automatic abstraction of a pushdown model from (possibly annotated) Java sources. We deal with the problems arising from the object-oriented nature of the Java programming language and we present some practical examples of properties which can be verified using our approach. A prototype tool called JAVACHECK was developed to perform this task.

The rest of the thesis is organized as follows: In Chapter 2, we define several useful notions related to model checking. We also give a brief overview of modal logics together with the corresponding complexity results. In the next chapter, we discuss the method of modeling Java programs using pushdown systems. Chapter 4 deals with the details of the implementation, especially with data structures and algorithms used. In

Chapter 5, we present a couple of examples of interesting properties which can be verified using our approach. In the last chapter we give a summary of obtained results, discussing possible extensions and related work.

Chapter 2

Basic Facts and Known Results

2.1 Transition systems

The structure of a modeled system can be expressed in many different formalisms (e.g. Petri nets or various process algebras). However, their semantics is often given in the terms of transition systems. Transition systems are especially advantageous for describing *reactive systems*, for which the denotational semantics is not so useful.

Definition 1 *A* labelled transition system *is a triple $\mathcal{T} = (S, Act, \longrightarrow)$, where:*

- *S is a set of* states*,*
- *Act is a set of* actions*, and*
- *$\longrightarrow \subseteq S \times Act \times S$ is a* transition relation*.*

Instead of $(s, a, t) \in \longrightarrow$ we write $s \xrightarrow{a} t$. We extend this notation to elements of Act^* in a natural way: $s \xrightarrow{w} t$ for $w \in Act^*$, $w = a_1 a_2 \ldots a_n$, if there exist states $s = s_0, s_1, s_2, \ldots, s_n = t \in S$ s.t. $\forall i. 0 \leq i < n : s_i \xrightarrow{a_{i+1}} s_{i+1}$.

Roughly speaking, a labelled transition system encodes the operational behaviour of a process. The set S represents all the states the system may enter, while Act describes the actions the process may perform. The $\longrightarrow$ relation then describes the actions available to states and their effect on the system. If we are interested only in the structure of a process, and not in the actions available at a given state, we can define the notion of an *(unlabelled) transition system*, which is a pair $\mathcal{T} = (S, \longrightarrow)$. All the related notions are defined analogously.

2.2 Pushdown systems

Pushdown systems (PDS[1]) are basically pushdown automata seen from a different point of view: we are not interested in the languages they recognize, but in the transition system they generate. These are infinite (in our case unlabelled) transition systems, having states of the form $\langle$control location, stack contents$\rangle$.

Definition 2 *A* pushdown system *(or PDS for short) 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 finite set of* transition rules*.*

A configuration *is a pair* $\langle p, w\rangle$*, where* $p \in P$ *is a control location and* $w \in \Gamma^*$ *is a* stack content*.*

What makes pushdown systems interesting from the practical point of view, is that they can be used for modeling programs with recursive procedures. This is done by using a configuration $\langle p, lw\rangle$ to represent the program being at a control point l, where w are return addresses of the calling procedures. Control location p can be used to hold any additional (global) information.

The class of *labelled* pushdown systems is denoted PDA here. In this chapter, we also mention two other classes of sequential systems. These are BPA (Basic Process Algebra) and FSA (Finite-State Automata) systems. BPA are basically PDA with just one control location, whereas FSA do not have a stack. We can, of course, also consider unlabelled versions of these classes of transition systems.

2.3 Temporal logics

Temporal logics are used to specify properties of processes. They can assert how the behaviour of a system changes over time; the formula may, for example, specify that *eventually* some state is reached, or that an error state is *never* reached.

In the following, we will assume that every computation of a transition system is non-terminating, i.e. for every state there is a transition leading from this state. Every transition system can be made infinite in this sense (e.g. by adding an infinite loop where necessary). Next we define some useful notions.

[1]We often the PDS abbreviation (for PushDown System) to denote the class of all pushdown systems. The actual meaning will be clear from the context.

Definition 3 *Let $\mathcal{S} = (S, Act, \longrightarrow)$ be a labelled transition system. A* run π *is an infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ such that every triple $s_i \xrightarrow{a_i} s_{i+1}$ belongs to the set of transitions. We also define the suffix $\pi^i = s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \cdots$ and we use $\pi(n)$ to denote the n-th state of the run π. For a state s let $runs(s)$ denote the set of all runs starting in s.*

There are two main classes of temporal logics which differ in their treatment of the branching structure of processes (i.e. the phenomenon of a nondeterministic choice). The truth of a *branching-time logic* formula depends on a state of the transition system and its whole computation subtree, whereas formulas of *linear-time logics* are interpreted over a single run of the system.

2.3.1 Linear-time logics

Formulas of linear-time logics are first interpreted over runs of a transition system. Formula φ holds at a state $s \in S$ (written $s \models \varphi$), iff all runs originating in s satisfy the formula.

LTL

Linear-Time Logic (LTL)[35] is a very popular logic widely used for verification of finite-state systems. The syntax and semantics is defined as follows:

$$\varphi ::= tt \mid \neg\varphi \mid \varphi \vee \psi \mid \langle a \rangle \varphi \mid \varphi \,\mathbf{U}\, \psi$$

$$\begin{aligned}
[\![tt]\!] &= runs(S) \\
[\![\neg\varphi]\!] &= runs(S) \smallsetminus [\![\varphi]\!] \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] \\
[\![\langle a \rangle \varphi]\!] &= \{\pi \mid first\ action\ of\ \pi = a \ \wedge\ \pi^1 \in [\![\varphi]\!]\} \\
[\![\varphi \,\mathbf{U}\, \psi]\!] &= \{\pi \mid \exists k \geq 0\ s.t.\ \pi^k \in [\![\psi]\!] \ \wedge\ \forall i.i < k :\ \pi^i \in [\![\varphi]\!]\}
\end{aligned}$$

A formula φ is true for a run π (denoted $\pi \models \varphi$) iff $\pi \in [\![\varphi]\!]$. We employ the following abbreviations:

$$\begin{aligned}
ff &= \neg tt \\
\varphi \wedge \psi &= \neg(\neg\varphi \vee \neg\psi), \\
\varphi \rightarrow \psi &= \neg\varphi \vee \psi \\
[a]\varphi &= \neg\langle a \rangle(\neg\varphi)
\end{aligned}$$

The additional two derived operators **F** and **G** (called "eventually φ" and "always φ") are often used:

$$\begin{aligned} [\![\mathbf{F}\varphi]\!] &= tt\,\mathbf{U}\,\varphi \\ [\![\mathbf{G}\varphi]\!] &= \neg\mathbf{F}\neg\varphi \end{aligned}$$

2.3.2 Branching-time logics

In branching-time logics, formulas are interpreted over states of a transition system. The least powerful branching-time logic is the Hennessy-Milner logic.

The Hennessy-Milner logic

Hennessy-Milner logic is a set of formulas defined as

$$\varphi ::= tt \mid \neg\varphi \mid \varphi \vee \psi \mid \langle a \rangle \varphi$$

where $a \in Act$. The semantics is defined as:

$$\begin{aligned} [\![tt]\!] &= S, \\ [\![\neg\varphi]\!] &= S \smallsetminus [\![\varphi]\!], \\ [\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!], \\ [\![\langle a \rangle \varphi]\!] &= \{s \mid \exists\, t \in S : s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]\} \end{aligned}$$

A formula φ holds in a state s (denoted by $s \models \varphi$) iff $s \in [\![\varphi]\!]$.

The logic UB

The logic UB ("uniform branching-time") is the extension of Hennessy-Milner logic by the operators **EF** and **EG**. Their semantics is defined as

$$\begin{aligned} [\![\mathbf{EF}\varphi]\!] &= \{s \in S \mid \exists \pi \in runs(s), \exists i \in \mathbb{N}\ s.t.\ \pi(i) \in [\![\varphi]\!]\} \\ [\![\mathbf{EG}\varphi]\!] &= \{s \in S \mid \exists \pi \in runs(s)\ s.t.\ \forall i : \pi(i) \in [\![\varphi]\!]\} \end{aligned}$$

Two other widely used derived operators **AF** and **AG** are defined as $\mathbf{AF}\varphi = \neg\mathbf{EG}\neg\varphi$ and $\mathbf{AG}\varphi = \neg\mathbf{EF}\neg\varphi$ respectively. The intuitive meaning of all these operators is summarised bellow:

$\mathbf{EF}\varphi$	"for at least one path will φ eventually hold"
$\mathbf{AF}\varphi$	"for all paths will φ eventually hold"
$\mathbf{EG}\varphi$	"there is a path on which φ holds forever"
$\mathbf{AG}\varphi$	"φ holds for all reachable states"

The Computation Tree Logic (CTL)

CTL is a very popular branching-time logic by Clarke and Emerson [10], and it is widely used for its simplicity and expressive power. CTL is the Hennessy-Milner logic extended by the (strong) existential until operator **EU**. The syntax and semantics is defined as follows:

$$\begin{aligned}[\![\mathbf{E}[\varphi\ \mathbf{U}\ \psi]]\!] &= \{s \mid \exists\pi \in runs(s), k \in \mathbb{N} : \\ &\qquad \pi(k) \in [\![\psi]\!] \wedge \forall i.i < k : \pi(i) \in [\![\varphi]\!]\}\end{aligned}$$

Notice that CTL subsumes the UB logic, as the operators **EF** and **EG** can be defined using the existential until operator :

$$\mathbf{EF}\varphi = \mathbf{E}[tt\ \mathbf{U}\ \varphi]$$

$$\mathbf{EG}\varphi = \neg\mathbf{EF}\neg\varphi$$

We also define an abbreviation $\mathbf{A}[\varphi\ \mathbf{U}\ \psi] = \neg\mathbf{E}[\neg\psi\ \mathbf{U}\ (\neg\varphi \wedge \neg\psi)] \wedge \neg\mathbf{EG}\neg\psi$.

The Modal μ-Calculus

The modal μ-calculus is a fixpoint logic of Kozen [29]. It is an extension of the Hennessy-Milner logic by variables and fixpoint operators.

Definition 4 *Let Var be a countable set of variables. The* modal μ-calculus *is a set of formulas defined by the syntax*

$$\varphi ::= X \mid \neg\varphi \mid \varphi \vee \psi \mid \langle a\rangle\varphi \mid \mu X.\varphi,$$

where $X \in Var$, $K \subseteq Act$ and in formulas of the form $\mu X.\varphi$, the variable X must occur under an even number of negations. In addition to the minimal fixpoint operator μ, *we also define the* maximal fixpoint operator ν *as an abbreviation*

$$\nu X.\varphi = \neg\mu X.\neg\varphi(\neg X)$$

where $\varphi(\neg X)$ is just a shorthand for replacing all free occurrences of X in φ by $\neg X$.

The semantics of the μ-calculus is defined with respect to a valuation of free variables. *Valuation* $\mathcal{V}$ is defined as a mapping $\mathcal{V} : Var \to 2^S$, assigning to every variable X the set of states which satisfy it. Valuation $\mathcal{V}[X := T]$, where $T \subseteq S$, is the same as the valuation $\mathcal{V}$ except for the variable X for which $\mathcal{V}[X := T](X) = T$.

Now we can define the semantics as follows:

$$\begin{aligned}[\![X]\!]_{\mathcal{V}} &= \mathcal{V}(X),\\ [\![\neg\varphi]\!]_{\mathcal{V}} &= S \setminus [\![\varphi]\!]_{\mathcal{V}},\\ [\![\varphi \vee \psi]\!]_{\mathcal{V}} &= [\![\varphi]\!]_{\mathcal{V}} \cup [\![\psi]\!]_{\mathcal{V}},\\ [\![\langle a\rangle\varphi]\!]_{\mathcal{V}} &= \{s \mid \exists\, t \in S : s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_{\mathcal{V}}\}\\ [\![\mu X.\varphi(X)]\!]_{\mathcal{V}} &= \bigcap\{T \subseteq S \mid [\![\varphi]\!]_{\mathcal{V}[X:=T]} \subseteq T\}\end{aligned}$$

The Alternation-free Modal μ-Calculus

The alternation-free modal μ-calculus [2] is a fragment of the general modal μ-calculus. While maintaining strong expressibility, it has better complexity properties when model checking certain classes of transition systems (e.g. FSA). To define it, we will utilise some notions:

Definition 5 *The set of formulas in* positive normal form *is defined by the following syntax:*

$$\varphi ::= X \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X.\varphi \mid \nu X.\varphi$$

It is straightforward to show that every formula is equivalent to some formula in the positive normal form. This is achieved by pushing negations inside and fixing the valuation where necessary.

Definition 6 *A μ-*formula *is a formula starting with the minimal fixpoint operator (i.e. of the $\mu X.\varphi$ shape). A ν-*formula *is defined analogously.*

Definition 7 *The* alternation-free modal μ-calculus *is defined as a fragment of the modal μ-calculus, where the following two restrictions hold:*

- *For every formula $\mu X.\varphi$, the variable X does not occur in any ν-subformula of φ.*
- *For every formula $\nu X.\varphi$, the variable X does not occur in any μ-subformula of φ.*

It remains to mention that CTL is a fragment of the alternation-free μ-calculus, the existential until operator can be as expressed as follows:

$$\mathbf{E}[\varphi\ \mathbf{U}\ \psi] = \mu X.(\psi \vee (\varphi \wedge \langle Act \rangle X))$$

2.3.3 Action-based vs state-based logics

All the logics defined in the previous sections were *action-based temporal logics*[3], interpreted on labelled transition systems. There is an alternative approach, where the information on top of arrows is ignored. Instead, formulas are built over atomic propositions and interpreted with respect to valuations which assign to every atomic proposition a set of states, where the proposition holds. Temporal logics of this kind are called *state-based temporal logics*.

[2] In this thesis also referred to as the μ_1-calculus

[3] With the exception of the modal mu-calculus, whose definition contains both the action-based and state-based features.

Example 1 (State-based LTL) *The syntax is defined as follows:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{U}\,\psi$$

given $p \in P$, where P is a set of atomic propositions. The semantics is then defined with respect to a valuation $\mathcal{V} : P \to 2^S$, and differs from its action-based counterpart only in treating of atomic propositions and of the relativised next operator $\boldsymbol{X}$ (which is often written as $\langle\rangle$):

$$\begin{aligned} [\![p]\!]_{\mathcal{V}} &= \{runs(s) \mid s \in \mathcal{V}(p)\} \\ [\![\mathbf{X}\varphi]\!]_{\mathcal{V}} &= \{\pi \mid \pi^1 \in [\![\varphi]\!]_{\mathcal{V}}\} \end{aligned}$$

There is, however, one important issue about state-based logics. The complexity and decidability of the model checking problem heavily depends on the atomic propositions used.

2.4 Known results on the model checking problem

Model checking is a technique for formal verification of transition systems. Formal definition follows.

Definition 8 (Model checking problem) *Given a formula φ of some temporal logic, transition system $\mathcal{T}$ and its state s, the* model checking problem *consists of deciding if s violates φ, that is whether there is some run of $\mathcal{T}$ starting at s that violates φ.*

2.4.1 Complexity results

The complexity of the model checking problem varies for different classes of transition systems and different temporal logics. There is, however, one common fact. The complexity of the model checking problem is usually much higher for the general case then for a fixed formula, as the model checking algorithm is often exponential only in the size of a formula, and not in the size of a transition system. This is reflected in the summary in Figure 2.1. The summary shows the complexity of both the general case (g) and for a fixed formula (ff). This table is taken from [36] and shows the complexity of the best known *deterministic* algorithms. Related results can be found in [30].

2.4.2 Available implementations

For finite-state systems (FSA), there is a broad variety of model checkers available, the most prominent being SPIN [23] or SMV [31] (using the LTL (CTL) logic), and CWB [32] (modal μ-calculus). To the best of our knowledge, there is no model checker available for BPA systems.

	FSA		BPA		PDA	
	g	ff	g	ff	g	ff
LTL	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$
CTL	$\mathcal{O}(n^k)$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$2^{\mathcal{O}(n^k)}$
μ_1-calculus	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$2^{\mathcal{O}(n^k)}$
μ-calculus	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$\mathcal{O}(n^k)$	$2^{\mathcal{O}(n^k)}$	$2^{\mathcal{O}(n^k)}$

Figure 2.1: The model checking complexity results

For pushdown systems, two model checkers appeared during the last year [15, 36]. The first of them uses the LTL logic, and is based on the algorithm given in [4] which was further improved in [15]. The model checker allows atomic propositions of the form "*the stack symbol s is currently on top of the stack*", whereas the algorithm given in [15] allows also the control locations to be taken into account. The algorithm used is polynomial in the number of states of pushdown system and singly exponential in the size of a formula. The JAVACHECK tool presented in Chapter 4 is intended to be used with this model checker.

The second tool, called ALFRED [36], is a model checker for PDS and alternation-free modal μ-calculus. It uses the algorithm described in [4] and arbitrary regular predicates on stack contents are allowed. The algorithm is singly exponential in the number of states of a pushdown system and in the size of a formula.

Chapter 3

Modeling Java Using PDS

In this chapter, we show how to model Java programs using pushdown systems (PDS). It should be mentioned that our model abstracts from data and concentrates only on modeling the control flow. We adopt an approach used in data-flow analysis and allow nondeterministic choice at the points where the control flow may branch (e.g. loops, conditional statements).

Using nondeterminism introduces *spurious paths* which do not correspond to any possible execution sequence of the modeled Java program. Nevertheless, we still have a safe *over-approximation* over the set of all possible executions. If the verified property is satisfied in some program model, then it is also satisfied in the original program. On the other hand, properties not satisfied by the model may be true for the original program, as the error trace is just a spurious path through our model.

3.1 Java overview

We start this chapter by giving a brief overview of the Java programming language. We will mention only the most important features relevant for purposes of our work. This overview is by no means complete and interested reader should refer to [27], which gives the full language specification. It should also be noted that we deal with Java 1.0[1], and do not consider any changes and extensions made in the following versions [2].

Java is an object-oriented language which originates from the C++. In fact, Java is much simpler than C++, because many features available in C++, which are not very "clean" in the sense of programming style, were removed. These include the `goto` statement, which was made unnecessary by introducing the labelled `break` and `continue` statements and sophisticated exception handling mechanism. Because Java is object-oriented, C

[1] The *current* version is Java 1.3.

[2] One of the most important of these extensions are the inner classes, which were introduced in Java 1.1.

constructs like `struct` and `union` were removed. Java also eliminates the operator overloading and multiple inheritance features of C++.

The most important simplification, however, is the absence of pointers. As Java does not have structures (and arrays are ordinary objects), pointers are no longer needed. Java also utilises automatic garbage collection, which substantially simplifies the memory management and makes programs significantly less error-prone. Except for the simple numeric, character, and boolean types, all the other types are ordinary objects. Then, a variable can be either of a basic type (e.g. an `int` value) or of a class type (a reference to an object of that type).

The Java *class hierarchy* is single-rooted and every class is inherited from exactly one superclass (with the exception of the root *Object* class). In addition, a class can implement one or more *interfaces*, which are simply collections of (abstract) methods for which the class must provide implementation. This effectively makes the multiple inheritance of C++ superfluous.

Files containing Java source code are called *compilation units*. There can be one or more classes per compilation unit and every compilation unit belongs to some *package*. There is a vast number of classes in the system packages that e.g. handle input and output (the `java.io` package) or create GUI components (the `java.awt` package).

Given source files, the Java compiler generates *bytecode*, which is stored in `class` files (one file per class). Java bytecode is an architecture neutral object file format. For bytecode to be executed, it must be *interpreted* by the *Java Virtual Machine (JVM)*. The list of platforms on which Java programs can be executed is then limited only by the fact, for which platforms is an implementation of Java Virtual Machine available.

3.2 Programs with procedures as PDS

Pushdown systems are especially suitable for modeling sequential programs with recursive procedures. The model we use establishes a relation between the control points of a program and corresponding configurations of the PDS generated. This is achieved by using the PDS stack to model a call stack of the program.

The PDS model is built in two steps. Firstly, a control flow graph is built for every method declared in the program. Nodes of the flow graph represent control points of the method and edges are labelled with the corresponding statements. Since we abstract from the values of variables, every choice is made nondeterministic. During the second phase, every method call in the flow graph is coupled with a start node of the called method.

At this point we can build the corresponding pushdown system. Assume we have a system of control flow graphs where N is a set of all control points. We construct a (single state) PDS $\mathcal{P} = (\{p\}, N, \hookrightarrow)$, which has

N as its stack alphabet. In this system, a configuration $\langle p, nw \rangle$ represents the program being in control point n, where w are return addresses of the calling procedures.

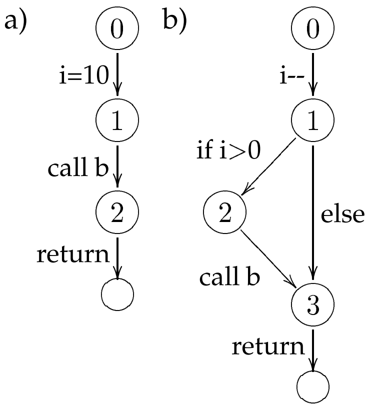
More formally, the transition relation $\hookrightarrow$ is defined by the following rules:

- $\langle p, n \rangle \hookrightarrow \langle p, n' \rangle$ iff there is an (n, n') edge in the corresponding control flow graph which is not a method call.
- $\langle p, n \rangle \hookrightarrow \langle p, mn' \rangle$ iff (n, n') is a method call edge, and m is an entry point of the method being called.
- $\langle p, n \rangle \hookrightarrow \langle p, \epsilon \rangle$ iff the edge leaving n contains a return statement.

The process of building a PDS from a Java source is depicted in Figure 3.1.

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
  i--;
  if (i>0)
    b(i);
}
```



$$
\begin{aligned}
\langle p, a_0 \rangle &\hookrightarrow \langle p, a_1 \rangle \\
\langle p, a_1 \rangle &\hookrightarrow \langle p, b_0\ a_2 \rangle \\
\langle p, a_2 \rangle &\hookrightarrow \langle p, \epsilon \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle p, b_0 \rangle &\hookrightarrow \langle p, b_1 \rangle \\
\langle p, b_1 \rangle &\hookrightarrow \langle p, b_2 \rangle \\
\langle p, b_1 \rangle &\hookrightarrow \langle p, b_3 \rangle \\
\langle p, b_2 \rangle &\hookrightarrow \langle p, b_0\ b_3 \rangle \\
\langle p, b_3 \rangle &\hookrightarrow \langle p, \epsilon \rangle
\end{aligned}
$$

Figure 3.1: Java program with the corresponding control flow graph and pushdown system.

The following sections of this chapter explain the construction in more detail. Section 3.3 describes the call graph construction, whereas Section 3.4 focuses on method call resolution – i.e. the process of linking method call sites to the corresponding method start nodes. Section 3.5 then discusses the possibility of using PDS to dynamically model the Java exception handling mechanism.

3.3 Intraprocedural analysis

This section deals with building a control flow graph of a method.

Definition 9 *We define a* control flow graph (CFG) *of a method as a directed graph* $\mathcal{G} = (N, E, n)$*, where* N *is a set of* control points *of the method,* E *is a set of (possibly labelled) edges, and* $n \in N$ *is a start node.*

The process of building a control flow graph is quite straightforward and well described in the literature. The only difficulty comes with the points where the control flow may branch. In Java, this problem is due to the `if-else`, `for`, `while`, `do-while` and `switch-case-default` constructs. The `try-catch-finally` construct must be treated in a different way than all the others, as the process of handling exceptions has both inter- and intraprocedural nature. We deal with this issue in Section 3.5.

It is important to notice that there can be an additional branching due to the `break` and `continue` statements. In Java, these statements can also be found in the version `continue` *label*/`break` *label*, where *label* is a label of some enclosing loop statement. There is a somewhat involved example in Figure 3.2 showing the principles mentioned.

3.4 Method call resolution

Having built a control flow graph, we must link the call sites (edges on which a method is called) to the initial nodes of control flow graphs of all methods which can be called from the given site.

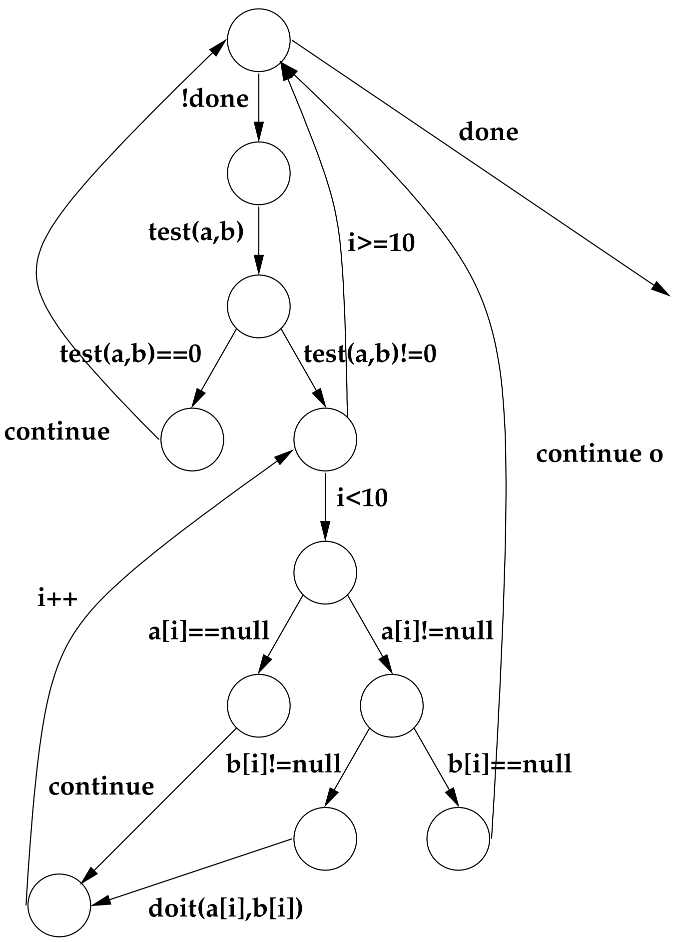
One important problem of static analysis of Java, as well as of other object-oriented languages, is that of statically determining what methods can be invoked by virtual method calls. It should be noted that in Java *late binding* is always used to find a method to be called.

Of course, virtual method resolution is not a new problem and was extensively studied for a variety of object-oriented languages [13, 18, 1, 39]. Main techniques used are *class hierarchy analysis* and *rapid type analysis*, and will be described in the following sections. This section is partially based on [39].

Terminology used A *method* is a function or procedure which is defined in a class and typically can access the internal state of an object of that class to perform some operation. It can be thought of as a procedure whose first parameter is an object to work on. This object is called the *receiver*. An exception are the Java static member functions which do not have a receiver, or *this* pointer. Finally a *message* is an invocation of a method.

```
o: while (!done) {
  if (test(a,b) == 0) continue;
  for (int i=0; i<10; i++) {
    if (a[i] == null)
       continue;
    else if (b[i] == null)
       continue o;
    doit(a[i], b[i]);
  }
}
```

(a) Program fragment



(b) The corresponding CFG

Figure 3.2: Control flow graph example

3.4.1 Class hierarchy analysis

Class hierarchy analysis (CHA) [13, 18] is a standard method for conservative (over-) estimating the run-time types of receivers. This is achieved by building an internal representation of the class inheritance hierarchy and then using this information to compute the run-time types of the receivers. The basic notion here is that of *Cone*.

Definition 10 *Given a receiver* o *of a declared class type* t*, we define the* Cone(t) *set as follows:*

- *If receiver* o *is of a declared class type* C*, the set* Cone(C) *consists exactly of C and all its subclasses.*
- *If receiver* o *is of declared interface type* I*, the set* Cone(I) *contains:*
  1. *The set of all classes that implement I or implement a subinterface of I (which we call* implements(I)*, and*
  2. *all subclasses of* implements(I).

Having computed the sets *Cone(C)* and *Cone(I)*, we proceed with the next step which is *call graph* construction. We start with a formal definition.

Definition 11 (Call graph) *A* call graph *is a set of nodes and directed edges. A call graph must include one node for each method that can be reached by a computation starting from the* `main` *method. Each node in the call graph also includes a collection of* call sites*. Edges in the call graph go from call sites within a call graph node to call graph nodes. The call graph must contain an edge for every possible calling relationship between call sites and call nodes.*

To make the notion of call graph more clear, see Figure 3.3 and 3.4. There is an example program together with its class hierarchy and the corresponding call graph. Each box in the class hierarchy graph stands for a single class together with the methods *declared* in this class. There are two call sites in the method `C.main`, one corresponding to the `a.m()` and one to the `b.m()` call.

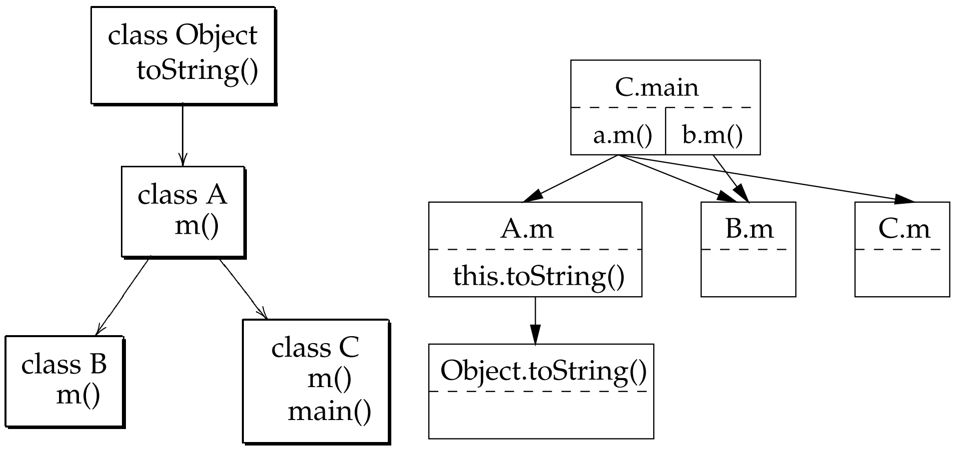
Call graph construction When building a call graph, we start with a set of nodes (one for each method). Given a call site of the form $o.m(a_1, \ldots, a_n)$, we find all possible targets of the call as follows. For each type C_i from the $Cone(o)$ set, we look up the class hierarchy until a class C' is found that declares a method $C'.m$ that matches the signature of m. The edge leading from the call site to $C'.m$ is then added.

We apply this construction to our program in Figure 3.3. For example, consider the `a.m()` call in the method `C.main`. Because `a` is declared to be

```
class A extends Object {
  String m() {
    return (this.toString());
  }
}

class B extends A {
  String m() { ... }
}

class C extends A {
  String m() { ... }
  public static void main(String[] args) {
    String s;
    A a = new A();
    B b = new B();

    ...

    s=a.m();
    s=b.m();
  }
}
```

Figure 3.3: Example program

of class type A, it can hold a reference to either an object of class type A, or to an object of one of its subtypes (B or C in this case). Therefore we must add three call edges leading to A.m, B.m and C.m from this call site. In contrast, the situation with the call site corresponding to b.m() is different. The declared type of b is B, which has no derived subclasses. So we can add only a single edge leading to B.m(). Now there is only one remaining unresolved call site, which is this.toString() in the method A.m. Although the declared type of this is A, an edge leading to Object.toString() is added, because this method is inherited and not overridden by the class A.



(a) Class hierarchy (b) Call graph

Figure 3.4: Call graph example

3.4.2 Rapid type analysis

Rapid type analysis (RTA) [1] starts with a graph generated during the class hierarchy analysis. This graph is then further reduced, using the information about all classes instantiated in our program.

The technique itself is based on a simple observation that a receiver can only have a type of an object instantiated using the new expression. Let $Instantiated$ denote the set of all object types instantiated in the program. Then given a receiver o of a declared type C, we use the set $Cone(C) \cap Instantiated$ as a better estimation of the set of all possible run-time types of o.

To demonstrate the functionality of RTA, let us return to the example in Figure 3.3. Assuming there are no other instantiations then those of a and b in the method C.main, the set $Instantiated$ has only two elements, A and B. Now consider the call site corresponding to the a.m() method call in the class `C`. Using rapid type analysis, only two edges leading from this call site are introduced, with `A.m` and `B.m` as their respective targets. That is because of the fact that `a` cannot hold a reference to an object of class `C`, as this class never gets instantiated.

3.5 Exception handling

This sections presents a possible way of modeling the Java exception handling mechanism. As the rules for handling the exceptions are very complex (see [27] for a thorough description), we only show the basic principles here. We start with a brief overview of Java exceptions.

3.5.1 Exceptions in Java

The Java language introduces exceptions as a way of handling semantic errors. An *exception* is a signal indicating that something bad happened, and it is an ordinary object in Java. Exceptions are *thrown* using the `throw` statement and the `try-catch-finally` construct is used to handle exceptions. `Try` establishes a block of code in which we check for possible exceptions. The `try` block is followed by one or more `catch` clauses, each handling a specific type of exceptions. The statements of a `finally` block are guaranteed to execute, regardless of how the code in the `try` block exits. The ordinary use of exceptions is depicted in Figure 3.5.

```
try { // code potentially raising IOException
  // throws FileNotFoundException,
  //   which is a subclass of IOException
  file = new java.io.FileReader("foo.bar");
  ...
  file.close();
} catch (IOException e) {
  // code recovering from IOException
  System.out.println("Error reading the foo.bar file.");
  System.out.println("Exception "+e+" signalled.");
} finally {
  // some finalisation code
  ...
}
```

Figure 3.5: Exception handling structures in Java

An exception is *caught* by a block of code, if it occurs in some `try` block and there is a `catch` clause whose argument is the same class or superclass as those of the exception. Then the associated block of code (*handler*) is executed and program proceeds further in a normal way. If an exception is not caught by the block of code that throws it, it propagates to the next enclosing block of code. If it is not caught anywhere in the method, it propagates to the invoking method, where it again propagates through the block structure. If it is not even handled in the `main` method, then the program aborts.

There are three basic exception types in Java: *checked exceptions*, *runtime exceptions* and *errors*, the last two being also called *unchecked exceptions*. Checked exceptions can only be thrown using the `throw` statement and every method must declare all the checked exceptions not caught in its body as a part of its signature. Checked exceptions are much easier to handle and because they are the most used, we would only concentrate on them. However, the exception handling mechanism, as described bellow, can be easily extended to handle also the unchecked exceptions, but only at cost of multiplying the size of the PDS generated.

3.5.2 Modeling the exception flow using PDS

We would like to extend our model to support the Java exception handling mechanism. To model Java exceptions, we make use of control locations. With every control location, we associate the information whether there is an exception being propagated and if so, of which type. It should be noted here, that the use of PDA states is crucial – it allows us to propagate exceptions through the method calls.

Assuming we have a PDS representation $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ of a Java program, we will show how to get a new PDS $\mathcal{P}' = (P', \Gamma', \hookrightarrow')$ which truthfully models Java exceptions. Let E be the set of all exception types thrown somewhere in the program. We define a new set of control locations $P' = \{(p, e) \mid p \in P \wedge e \in (E \cup \{0\})\}$. The second component of a (p, e) pair is an *exception flag*. The exception flag e means that an exception of type e was thrown, and 0 that no exception is currently being propagated. This implies that every transition $\langle p, \alpha \rangle \hookrightarrow \langle q, \beta \rangle$ is substituted by the $\langle (p, 0), \alpha \rangle \hookrightarrow' \langle (q, 0), \beta \rangle$ transition. For simplicity, we will only show the exception flag component of the state.

We also have to extend our set Γ of stack symbols. This is necessary to accommodate handling of the `finally` construct. We introduce a new special stack symbol f_e associated with every `try` block. In addition, we require that $E \subseteq \Gamma'$. This symbol will hold information about the exception currently being handled by the `finally` block.

In our treatment, we make the following assumptions:

- There is a single exit terminal node for the `try` and `catch` and `finally` CFGs. This could be easily satisfied, if there are no `break`,`continue` and `exit` statements present. The handling of these statements will be described separately.

- There is a `finally` clause associated with every `try` block. If this is not the case, we can simply add a void `finally` clause with no effect on the program semantics.

Modeling a `try` block

The `throw` statement Let t_i be the CFG node corresponding to a `throw` statement, and e the type of the exception being thrown. Then there are two possible cases:

- e is caught by the ith `catch` clause. In this case, we add a transition $\langle 0, t_i \rangle \hookrightarrow \langle 0, c_{i,1} \rangle$. Notice, that exception flag is cleared after the transition. This corresponds to the fact, that no exception was thrown in the `catch` block.

- e is not caught. Finally block is then going to be executed in exceptional mode, the reason being the exception of type e. The corresponding transition is $\langle 0, t_i \rangle \hookrightarrow \langle 0, f_1 e f_e \rangle$.

Normal completion If the block completes normally, then the control is in node t_ϵ and the `finally` block is to be called in normal mode. The corresponding transition is $\langle 0, t_\epsilon \rangle \hookrightarrow \langle 0, f_1 0 f_e \rangle$.

Modeling a `catch` block

The `throw` statement Let c_i be a CFG node corresponding to a `throw` statement, and e the type of the exception being thrown. Then the `finally` block is going to be called in exceptional mode, and thus we add a transition $\langle 0, c_i \rangle \hookrightarrow \langle 0, f_1 e f_e \rangle$

Normal completion If the `catch` block successfully handles the exception, then control is in node c_ϵ, and we call the corresponding `finally` block in normal mode: $\langle 0, c_\epsilon \rangle \hookrightarrow \langle 0, f_1 0 f_e \rangle$.

Modeling a `finally` block

As we could see earlier, the top of the stack inside the `finally` block is of the shape $\boxed{f_i e f_e \ldots}$ if `finally` was called in exceptional mode and $\boxed{f_i e f_e \ldots}$ otherwise. When the control is about to leave the `finally` block, it simply pops the top stack symbol. The next behaviour depends on both the state and stack symbol. There are two possibilities:

Exceptional exit Exception was raised in the `try-catch-finally` block and not handled there. There three possibilities of this:

a) **the `finally` block was called in exceptional mode**
Then exception flag is 0 and e is on top of the stack. We introduce a set of transitions $T_1 = \{\langle 0, e\rangle \hookrightarrow \langle e, \epsilon\rangle \mid e \in E\}$. This transitions are common for all the methods.

b) **an exception was thrown within the `finally` block**
Then exception flag is e and 0 is on top of the stack. As in previous state we introduce a set of transitions $T_2 = \{\langle e, 0\rangle \hookrightarrow \langle e, \epsilon\rangle \mid e \in E\}$, also common to all the methods.

c) **both a) and b)**
Both the exception flag and top of the stack contain an exception type. In this case, the more recent reason is used. This implies a family of transitions $T_3 = \{\langle e_1, e_2\rangle \hookrightarrow \langle e_1, \epsilon\rangle \mid e \in E\}$.

As the reader could see, in all three cases we end up in a configuration $\langle e, f_e \ldots\rangle$. The behaviour now depends on whether there is a lexically enclosing `try` block.

- The `try` block is present. Then transitions are added as if the exception of type e was thrown by the `throw` statement in the enclosing in this `try` block.

- The `try` block is not present. Then the behaviour is the same as as if the exception of type e was thrown by the `throw` statement not enclosed by a `try` block.

Normal completion Firstly, we must pop the stack using the $\langle 0, 0\rangle \hookrightarrow \langle 0, \epsilon\rangle$ transition. Then we continue with the node (say n) immediately following the `try-catch-finally` construct. The corresponding transition is $\langle 0, f_e\rangle \hookrightarrow \langle 0, n\rangle$.

Modeling exception propagation through method calls

As was mentioned earlier, exception flag is used to propagate exception through method calls.

`throw` statement outside a `try` block When such a `throw` is reached the top stack symbol is removed from the stack and the exception is left to be handled by the calling method. More formally, let i be the CFG node corresponding to a `throw` statement, and e the type of the exception being thrown. Then we add a transition $\langle 0, i\rangle \hookrightarrow \langle e, \epsilon\rangle$.

Modeling a method call When the called method returns, we must distinguish, whether it completed normally or abruptly. For this reason, we introduce a whole set of new transitions for every method call site. These will transfer the control to the corresponding `catch /finally` clauses when necessary.

More formally, let n' be node immediately following the method call (this node is on top of the stack when the called method terminates). For every exception type $e \in E$ we introduce a transition $\langle e, n' \rangle \hookrightarrow \sigma$, where σ is a target of a transition corresponding to the `throw e` statement, would such transition appear at n'.

Handling of the `return`, `break`, and `continue` statements

When handling exceptions, we must be aware of the following fact. If control leaves the `try` block as a result of executing one of the `return`, `break` or `continue` statements (also referred to as *control transfer statements*, then the associated `finally` block must be executed before the control transfers to a new destination. Moreover, the above mentioned statements can also appear in the `catch` and `finally` blocks.

Handling of the control transfer statements is very similar to handling exceptions and can be done by introducing a new exception type for every such a statement. New transitions are added as for the "normal" exception types, with these modifications:

- When a f_e symbol is on a top of the stack, the transition is made according to the semantics of the present exception type.
- Exception types corresponding to the control transfer statements do not propagate through method calls.

Chapter 4

The JAVACHECK Tool

To demonstrate a practical usefulness of the previously introduced formal model of Java programs, a prototype tool called JAVACHECK was designed. In this tool we implemented several concepts from the previous chapter and, in addition, we show how the model abstraction tool can help the user in various phases of the verification process, e.g. in interfacing with the model checker.

Designing JAVACHECK, we have chosen Java as the implementation language and this choice proved to be very advantageous. As an object oriented language, Java allows to clearly express the concepts behind the program structures and algorithms. The often discussed run-time efficiency was not an issue, as the model abstraction is not a time-critical task. Moreover, the use of Java allowed us to take the advantage of dynamic class loading mechanism in the implementation of static class analysis. All the code of JAVACHECK is stored in the `javacheck` package and its subpackages.

JAVACHECK is intended to be used together with the model checker described in [15]. Therefore the PDS and formula format of JAVACHECK is compatible with this tool. However, it is easy to modify our tool to interface with other model checkers for PDS by making only a few minor changes, mainly in the output routines. For example, we have successfully modified JAVACHECK to work with the ALFRED model checker of Polanský [36].

In the rest of this chapter we describe JAVACHECK in more detail, focusing on several different implementation aspects. We intend to describe only the crucial ideas here and the reader can find a more detailed information on the classes, methods, and attributes used in the generated documentation available on the accompanying CD.

4.1 The JAVACHECK overview

The JAVACHECK tool builds an abstract PDS model of a Java program – taking a list of Java source files, it produces a PDS representation which is then used as an input for the model checker. In addition, several other features are implemented. These include a possibility of annotating the source code with calls to static methods of the `javacheck.Action` class. All methods of this class have an empty body (and therefore they have no effect on the program execution), but allow the user to control the process of model abstraction.

The JAVACHECK tool works in two main phases. During the first phase, (intraprocedural) control flow graphs of all methods are built. The code implementing this phase is built around the CUP LALR parser generator [24], which in turn makes use of the JFLEX lexical analyser generator [28]. The generated scanner together with several helper classes is stored in the `javacheck.parser` package.

In the second phase, a PDS representation is built. This phase is responsible for method call resolution and for linking the call sites to all possible method entry points available at the given call site. In addition, a visual representation of control flow graphs, class inheritance graph, as well as some internal data structures can be printed when requested by the user.

4.1.1 Data structures used

The information obtained from the source files is stored in objects corresponding to the following input entities:

- *Compilation units* are represented by objects of the `javacheck.JUnit` class.
- *Classes* are represented by objects of the `javacheck.JClass` class.
- *Methods* are represented by objects of the `javacheck.JMethod` class.

Every class, method, and control point is assigned a unique id-number. Each control point can then be uniquely identified using three numbers. This is reflected in names of stack symbols. A stack symbol $\mathtt{s}xxx\_yyy\_zzz$ corresponds to the zzzth control point in the yyyth method of the xxxth class.

The control flow graph of a method is represented as a set of directed edges between control points. Every edge is implemented as an instance of the `javacheck.Edge` class. To make things easier, a method is identified using only the fully qualified class name, the method name, and the number of parameters. The notation used is `java.lang.Object.clone:0`.

4.1.2 JavaCheck limitations

JavaCheck currently does not handle the full Java 1.3, as described in [27]. This is not a big problem, as JavaCheck supports a reasonable subset of Java and serves well as a "proof of a concept" tool. To be precise, it still suffers from the following limitations:

- Nested classes, as well as other features not present in Java 1.0 are not handled. When a nested class is encountered, a warning is printed.
- The exception handling mechanism, as described in Section 3.5, is not implemented.
- Methods with the same name and the same number of parameters (but of different types) are not distinguished.
- No support for interfaces is included.

4.1.3 Performance issues

The run-time complexity of the model checking algorithms depends on the size of a PDS given on input. This size is proportional to the number of nodes of all method control flow graphs (as the number of stack symbols equals to the number of CFG nodes). We have run JavaCheck on several different Java programs to see how the number of nodes changes with the number of lines of the input. For most programs, the ratio varies from $1:3$ to $1:4$.

However, the number of stack symbols can usually be further reduced. This can be achieved by removing those edges and nodes in the control flow graph, which have no effect on the verified property. JavaCheck currently does not use any such control flow graph optimisation. It would be interesting to implement some and see the impact on the size of the generated PDS.

4.2 Implementation details

4.2.1 Method invocation resolution

When building a method control flow graph, we are interested in the fact, of which declared type is the object on which the method is invoked. In other words, for every method invocation expression we want to get a fully qualified name of the method, which can later be linked to the corresponding declarations.

This task is not so easy in Java, because method invocation expressions can be quite complicated. This is caused by the fact that all local variables, field names, and other method invocations can occur in method invocation expressions. To be able to perform the method invocation resolution effectively during the first phase, we need the following information:

- Which *local variables* are currently visible and what are their declared types. This information is dynamic and depends on the lexical structure of the already parsed portion of the source code.
- The list of all *class members* (methods and fields) referenced in the program, together with their *function prototype/declared type*. It should be pointed out, that there is no restriction on where the class member declaration must appear in the source with respect to the place of its first use. So in Java we can freely reference a field or call a method before it is declared.

Here we present possible solutions for the two problems.

Local variables handling

To keep a track of local variables, we use a dynamic stack-like structure. There are two types of objects which can be pushed on stack: 1) *variable records* and 2) *new scope marks*. Each variable record consists of a variable name and of a fully qualified type name associated with this variable. An appropriate variable record is pushed on the stack when a variable declaration is encountered. When entering a nested lexical scope, a new scope mark is pushed on the stack. At the end of the block, all variable records above the last scope mark are deleted, as well as this mark. *Searching* is implemented quite trivially: Having a variable name, we search the stack in a top-down manner and the first record with a matching name found is that of this variable.

Class member resolution

We need to know the type of all class members (fields and methods) used in the program. The first possibility is to get the necessary information using a new compiler pass. During this pass, all the given compilation units are processed and the information about fields and methods is retrieved. This approach is straightforward and easy to implement, but there is a problem in case we do not have the sources for all classes used in the program (e.g. for commercial libraries). In this case, we are not able to resolve those method invocation expressions which contain a reference to a member of such a class. This limitation can be overcome using the ideas shown in the alternative method resolution approach presented bellow.

The solution used in JAVACHECK is much less effective, but it does not require a separate compiler pass. Instead, the information stored in `class` files is used. We make use of the fact that compiled `class` files include (in addition to bytecode) also the following information:

- The total number and types of formal parameters and a return type for every method *declared* in the class.
- The type of every field *declared* in the class.

The restriction imposed by this approach is that it is necessary to have a compiled `class` file for every compilation unit used. As we actually have the source files, this should not be an issue. The possibility of checking Java programs having only the compiled `class` files is discussed later in Section 6.1.

4.2.2 Class hierarchy analysis

Having a control flow graph built for every method, we want to link call sites to the entry points of the methods which can be invoked on the object associated with the call site. We start with a construction of a *total call graph*. In this graph, we associate with *every* class type c and *every* method m available in this class (declared or inherited) all corresponding graph nodes. This information is used in the second phase, when a method m is invoked on an object of a declared class type c.

In the following, we use $E_{c:m}$ to denote the entry point of a method m declared in a class c. (Note that inherited methods which are not redeclared in the class have no entry points.) Now with every class type c and method m available in this class (either declared or inherited) we associate a set $M_{c:m}$ of method entry points. The $M_{c:m}$ set contains entry points of all methods which can be called when a method m is invoked on an object of a declared class type c. The set is constructed as follows:

- $E_{s:m} \in M_{c:m}$ if the method m is inherited from a superclass s and not redeclared in the class c.
- $E_{d:m} \in M_{c:m}$ if the method m is declared in a class d and $d \in Cone(c)$.

We start with building a class hierarchy graph, whose nodes (representing classes) contain the available methods and for the declared methods their entry points. A single depth-first class hierarchy tree walk is used to construct the $M_{c:m}$ sets. These sets are then stored in the class hierarchy together with the class representation and used during the second phase of building the PDS.

4.2.3 Finite automata support

The JAVACHECK tool allows the user to explicitly use control locations for keeping run-time information. With every PDS, a deterministic finite automaton (FA) can be associated, whose states are the states of the resulting

PDS. The FA definition file can be given as an optional argument on the command line. If no automaton is given, the generated PDS will have only one state `p` which has no special meaning and will be used in all transitions. Otherwise, the number of transitions is multiplied by the number of FA states as the construction described in Section 3.2 does not use states (we do not model the exception flow). Therefore we enable every transition in every state of the PDS.

The only exception are the transitions corresponding to invocations of the `javacheck.Action.faMove` method. This method has a single argument, which is the label of a transition to be made. Assuming that a PDS is in some state p, method `javacheck.Action.faMove("a")` is called, and there is a $p \xrightarrow{a} q$ transition in the associated FA, then the state of PDS is changed to q. For the construction to be unambiguous, we require the FA transition relation to be total.

The format of the finite automaton definition file is as follows:

initial_state
state label state / 1st transition */*
...
state label state / last transition */*

4.2.4 Interfacing with the model checker

As described in Section 2.4.2, the only atomic predicates available in formulas are of the form `pred`, where `pred` is a stack symbol. This predicate holds for all configurations with the symbol `pred` being on top of the stack. Using stack symbols in formulas is not desirable, as we want to create formulas independently of the generated PDS.

JAVACHECK can help the user in several ways. First, given a file containing LTL formulas with atomic predicates of the form `m:n`, where `m` is a fully qualified name of a method and n is the number of formal parameters, JAVACHECK automatically substitutes each such name with the stack symbol corresponding to the entry point of the method `m:n`. Thus, the proposition `m:n` is true whenever the method `m:n` is called.

Obviously, we may want to express properties dependent on general control points, not only on the entry points of procedures. To facilitate this, the user can annotate the program with so-called *watchpoints*. A watchpoint is defined by inserting a call to the `javacheck.Action.watch` method. The only argument of this method is a unique integer id-number. In a formula, an atomic predicate `@n` corresponds to the watchpoint `n` being on top of the stack. A predicate of this kind is automatically substituted by the stack symbol corresponding to the control point from where the `javacheck.Action.watch(n)` method is invoked.

The name of the file containing LTL formulas to be substituted is specified using the `-f file` argument (optional). The syntax of LTL formulas is the same as in the SPIN model checker and is given in Figure 4.1. The only differences from the LTL syntax given in Section 2.3.1 are that the **F** and **G** operators are written as <> and `[]`, and **V** is a dual operator to **U** (i.e. $\varphi\, \mathbf{V}\, \psi \equiv \neg(\neg\varphi\, \mathbf{U}\, \neg\psi)$). The boolean connectives $\neg$, $\wedge$, and $\vee$ are written as !, $\&\&$, and $\|$ respectively.

$$\begin{array}{rcl} formula & ::= & \texttt{true} \mid \texttt{false} \mid un\_op\ formula \mid formula\ bin\_op\ formula \mid \\ & & atomic \\ atomic & ::= & method\_identifier \mid @number \\ un\_op & ::= & \texttt{!} \mid \texttt{X} \mid \texttt{<>} \mid \texttt{[]} \\ bin\_op & ::= & \texttt{U} \mid \texttt{V} \mid \texttt{\&\&} \mid \| \mid \texttt{<->} \mid \texttt{->} \end{array}$$

Figure 4.1: The LTL formula syntax

4.2.5 Control flow graph visualisation

JAVACHECK allows the user to display the generated control flow graphs. If `-d` option is used, then the generated graphs are stored in the `dots` subdirectory of the current directory. A new directory is created when necessary. The file-naming convention is that the control flow graph for a method `m` of two arguments declared in the class `c` of package `p` is stored in the file `p#c#m:2.dot`.

Control flow graphs are stored in the `dot` format, which is a native format of DOT. DOT is a graph layout tool developed by Stephen North and colleagues at AT&T as a part of the GRAPHVIZ package (see [37]). Using DOT, a `dot` file is converted to a graph in PostScript format.

4.3 Usage and options

Assuming the JAVACHECK classes are stored in a directory on the CLASSPATH, the program can be executed by the following command:

```
java javacheck.Main [options] file [file ...]
```

Command line options:

- `-a file` Reads a finite automaton definition from *file*. The FA syntax is described in Section 4.2.3.
- `-b method` Sets the initial method of a program. If omitted, the `main:1` method of the first input file is used instead. An error will occur if there is no such a method.

`-d` Generates the `.dot` files for all declared methods. These are stored in the `dots` subdirectory of the current working directory. For further information on `dot` files, see Section 4.2.5.

`-f file` Substitutes all formulas in *file*. The substitution algorithm as well as formula syntax is described in Section 4.2.4.

`-i` Makes all the runs infinite, adding an infinite loop when necessary. It is important to use this option when the program has a finite run, because otherwise the model checking algorithm would not function properly.

`-o file` The generated PDS model will be stored in *file*. If omitted, the `out.pda` file in the current directory will be used.

`-s` Prints a summary. This includes class hierarchy, and a list of all compilation units, classes, and methods together with their assigned id's.

`-v` Toggles the verbose output on. The JAVACHECK tool will produce a lot of information about the progress of the model abstraction process.

Chapter 5

Examples

In this chapter, we present several examples of properties which can be verified using our model.

5.1 A graph plotting example

This example, which is taken from [15], shows how a complex call dependency between methods can be verified. Look at the program in Figure 5.1. This is a program for plotting random bar graphs using the commands `goUp`, `goRight`, and `goDown`. One of the correctness properties is the requirement that an upward movement will not be immediately followed by a downward movement and vice versa.

The desired property can now be specified using the following formula:

$$\begin{aligned}\varphi &= \mathbf{G}(goUp \rightarrow ((\neg goDown\ \mathbf{U}\ goRight) \vee \mathbf{G}\neg goDown)) \\ &\wedge \ \mathbf{G}(goDown \rightarrow ((\neg goUp\ \mathbf{U}\ goRight) \vee \mathbf{G}\neg goUp))\end{aligned}$$

where the atomic predicates $goUp$, $goRight$, and $goDown$ correspond to the `goUp()`, `goRight()`, and `goDown()` methods being called (i.e. their corresponding entry points are on top of the call stack). This formula evaluates to true in our system confirming that the correctness assumption is satisfied.

5.2 A generic counter

Sometimes, it can be useful to check whether some action occurs at most/at least n times. For example, if we have a buffer which can hold only four entries, we must assure that we will not demand more. As the program can be considerably complex, this property is not easy to check.

```
static void m(){
  double d = java.lang.Math.random();
  if (d < 0.66) {
    s();
    goRight();
    if (d < 0.33)
      m();
  } else {
    goUp();
    m();
    goDown();
  }
}

static void s() {
  if (java.lang.Math.random() < 0.5)
    return;
  goUp();
  m();
  goDown();
}

public static void main(String args[]) {
  s();
}
```

Figure 5.1: The graph plotting program

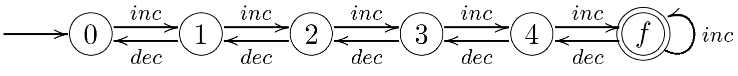


Figure 5.2: The FA counter

A solution to this problem is to associate a FA which works as a counter with the PDS model of the program. A finite automaton which implements the $\langle 0, 4 \rangle$ counter is shown in Figure 5.2. The program is then annotated using the `javacheck.Action.faMove("inc")/faMove("dec")` method calls on those places where a buffer entry is inserted/removed. Now it remains to check whether any configuration with the control location f is reachable, which can be easily done using the **F** operator.

5.3 Application to Java security

The Java Development Kit (JDK) 1.2 introduced a completely new security model (see [20] for a detailed description), which is based on the notion of *protection domains*. Every Java class is assigned to some protection domain, and every protection domain is associated with a set of *permissions*. Permissions are verified using the `AccessController` class from the standard library. The `CheckPermission` method of this class checks whether a given permission is granted and throws an exception if not.

For a permission to be granted, all methods on the call stack must have this permission granted OR a caller whose domain is granted the said permission has been marked as *privileged* and all of the methods subsequently called by this caller have the permission. In other words, methods on the call stack are checked in a top-down manner, until either the stack bottom or a privileged caller is reached. A caller is marked as privileged when it calls the `doPrivileged` method [1]. This method takes as an argument an object of the `PrivilegedAction` class, whose only method `run()` contains the code to be made privileged. In all the examples we have seen, this is done using an anonymous inner class. The privileged call can therefore be identified statically.

Formalisation in our model Having built a PDS representation of a Java program, we can identify the set of permissions associated with every stack symbol (which in turn corresponds to some control point of the program). As we said earlier, we also can statically identify the set of stack symbols which have been marked privileged. Having this information, we can describe the interesting call stack structures (e.g. those violating some security property) using regular predicates on the stack contents.

5.3.1 Electronic commerce example

In this section, we will illustrate the model through an electronic commerce example, which is taken from [26]. There are four protection domains in-

[1] In the first releases of JDK 1.2, the `beginPrivileged/endPrivileged` method pair was used instead of the `doPrivileged` method to designate privileged code.

volved, called *System*, *Provider*, *Client*, and *Unknown*:

- The system (Figure 5.3) provides code to implement a controlled floating point variable. This variable can be accessed only using the `read` and `write` methods, which check the respective `Read` and `Write` permissions. The *System* domain also contains the code necessary to launch the whole application (not shown here).
- The service provider provides an account manager (Figure 5.4), which uses the controlled variable. This manager has two methods – a transaction `debit` and a boolean query `canpay`. These two methods can `read` and `write` in privileged mode, as they can be called by clients which are not granted the `Read` and `Write` permissions.
- Finally, there is a client (Figure 5.5) which can operate with the account (only the relevant code is shown).
- Moreover, to illustrate the handling of illegal code, an intruder (Figure 5.6) is introduced, who tries to execute unauthorised operations.

The permissions are associated with the protection domains as given in the Table 5.1.

	`Debit`	`Canpay`	`Read`	`Write`
$System$	YES	YES	YES	YES
$Provider$	YES	YES	YES	YES
$Client$	YES	YES	–	–
$Unknown$	–	–	–	–

Table 5.1: The permissions associated with each permission domain

Verification of security properties

As a global statement about the security of the system, we state that all the calls leading to a modification of the balance must posses the `Debit` permission, and all calls leading to disclosure of the balance must posses the `Canpay` permission. We formalize this fact in our model.

At first, we build a PDS representation of the system. Then we can describe all the call stacks that have the `Debit` permission using the (regular) atomic predicate P_{Debit} and similarly all the call stacks that have the `Canpay` permission using the predicate P_{Canpay}. The (once again, regular) atomic predicate P_{Read} is assumed to be true at the point after the check in the `read` method, and the predicate P_{Write} is defined in the same manner

```
public class ControlledVar {
  private float var;
  public void write(float newVal) {
    AccessController.checkPermission(Write);
    var = newVal;
  }
  public float read() {
    AccessController.checkPermission(Read);
    return var;
  }
}
```

Figure 5.3: The system code (*System* domain)

```
public class AccountMan {
  private ControlledVar balance;

  public boolean canpay(float amount) {
    AccessController.checkPermission(Canpay);
    boolean res = (boolean) AccessCon-
troller.doPrivileged (
      new PrivilegedAction() {
        public Object run() {
          return balance.read() > am;
        }
      });
    return res;
  }

  public void debit(float amount) {
    AccessController.checkPermission(Debit);
    if (this.canpay(amount)) {
      AccessController.doPrivileged ( new Privi-
legedAction() {
        public Object run() {
          balance.write(balance.read() - am);
        }
      });
    } else ...
  }
}
```

Figure 5.4: The account manager code (*Provider* domain)

```
public void spender() {
  float spend = 10000;
  if (account.canpay(spend)) {
    account.debit(spend);
  }
  spender();
}
```

Figure 5.5: The application code (*Client* domain)

```
public void clyde() {
  account.debit(5000000);
  clyde();
}
```

Figure 5.6: An uncertified application (*Unknown* domain)

for the `write` method. We can express the mentioned security statements using the following formula:

$$\varphi = \mathbf{G}(P_{Write} \rightarrow P_{Debit}) \wedge \mathbf{G}(P_{Read} \rightarrow P_{Canpay})$$

This formula evaluates to true in our system. A closer look reveals that the program is too defensive. It can be optimized by removing the check in the `canpay` method, as there is no path leading to this method passing through the code without the `Canpay` permission. For a similar reason, the checks in `read` and `write` can be removed. We can now verify the modified program to see that φ still holds, which confirms the correctness of our optimisations. A similar analysis shows that the check in `debit` is really necessary, as the uncertified application possesses a link to the `debit` method.

Chapter 6

Conclusion, Possible Extensions and Related Work

We have investigated the possibility of automatic PDS model abstraction of Java programs. We have given an algorithm for translating Java programs into PDS and discussed the related issues. A software tool called JAVACHECK was implemented to prove our concepts. Finally, we presented a couple of examples of properties, which can be verified using our model. Some possible extensions to our model are discussed in the following section and we close the thesis giving an overview of the related work.

6.1 Possible extensions

Modeling data In this paper, we have focused only on modeling the control flow of a program. To make the analysis more accurate and powerful, the data flow must be taken into account. There is a natural way of storing data in pushdown systems. Stack symbols and control locations can be extended to hold global (control locations) and local (stack symbols) data. This approach is followed in [17], where binary data diagrams (BDDs) are used to symbolically represent the underlying PDS. The use of BDDs allows the representation to be exponentially smaller comparing to the explicit PDS description.

As the variable domains are very large, data abstraction techniques must be used to make models tractable. This is usually achieved by restricting a large data type domain to a small set of pseudo-values (e.g. positive/zero/negative for a numeric type). *Predicate abstraction*, introduced by Graf and Saidi [21], is a popular form of over-approximation. The basic idea is to replace a concrete variable by a boolean variable that evaluates to a boolean formula (predicate) over the original variable. This concept is easily extended to handle multiple predicates and predicates over multiple variables. For an example of use of predicate abstraction in an object-oriented language, see e.g. [42].

Bytecode Verification Instead of verifying a program given its source code, we can start with the compiled bytecode. This makes the program analysis much simpler, as bytecode has quite a restricted set of basic statements. Even in the case the bytecode have been optimised, there is still a Java source code line number associated with every statement, allowing the generated error trace to be matched with statements of a program source code. The advantage of this approach is that we can verify even those parts of program (libraries etc.) for which we do not even have source code available.

As bytecode is a stack based code, the analysis is not quite straightforward. This obstacle can be overcome using the SOOT compiler framework developed by the Sable group at the University of McGill [40]. Using SOOT, Java programs are transformed into an intermediate JIMPLE representation (one of intermediate representations available in SOOT). JIMPLE is a typed, 3-address representation of bytecode, where stack is eliminated and replaced by additional local variables. Another advantage is that we can employ sophisticated techniques for virtual method call resolution, which were implemented as a part of the SOOT framework [39].

6.2 Related work

In recent years, a few attempts have been made to automatically model check programs written in higher programming languages. As far as we know, only model checking of finite-state Java models has been attempted and tools such as JCAT [25] and JAVA PATHFINDER [22] were built. These tools translate a Java program directly into a relatively expressive PROMELA language, the input language of the SPIN [23] model checker. Both the tools support concurrent thread execution and allow the user to annotate programs with various assertions. The weakness of these tools is their inability to significantly compress the PROMELA model based on the property to be checked, so model checking for nontrivial programs easily becomes intractable.

The BANDERA tool-set [12], which utilises the SOOT framework, overcomes some of these limitations and is, recently, the most sophisticated tool available. BANDERA employs several abstraction techniques which restrict variable domains and make the resulting model significantly smaller. The generated model is encoded in an intermediate language representation, which is then converted to the input language of some available model checker (e.g. SPIN [23] or SMV [31]).

In the area of model checking of infinite-state systems, there are only few existing attempts to verify programs written in widely used languages as C or Java. The SLAM toolkit [3] serves for verification of C programs and has been used to validate security properties of Windows NT device drivers. In SLAM, C programs are translated into a model called *boolean programs*, which are basically C programs where all variables have boolean type. The process of model abstraction is highly automated and models are automatically refined when necessary. A symbolic model checker BEBOP [2], which is used to verify boolean programs, represents sets of states implicitly using BDDs and uses an interprocedural data-flow analysis algorithm to solve the reachability problem.

In [17], another model checker for boolean programs is presented. In contrast with BEBOP, this model checker can deal with liveness and fairness properties, as arbitrary LTL properties are allowed. The model checker works with symbolic pushdown systems (SPDS), a compact representation of the pushdown systems studied in [15]. As it is further shown in [17], this approach leads to some efficiency advantages over the BEBOP tool.

Bibliography

[1] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96*, pages 324–341, 1996.

[2] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN' 00: Model Checking of Software*, 2000.

[3] T. Ball and S. K. Rajamani. Checking temporal properties of software with boolean programs. In *Workshop on Advances in Verification (with CAV 2000)*, 2000.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170.

[7] O. Burkart and B. Steffen. Composition, decomposition and model checking optimal of pushdown processes. *Nordic Journal of Computer Science*, 1995.

[8] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429, 1997.

[9] E. Clarke. Symbolic model checking: 10^{130} states and beyond. In *Proceedings of the 11^{th} CADE*, volume 607 of *LNAI*. Springer Verlag, 1992.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*, 1981.

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

[12] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Verlag, 1995.

[14] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard university, 1981.

[15] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV: International Conference on Computer Aided Verification*, 2000.

[16] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural dataflow analysis. In W. Thomas, editor, *Proceedings of FOSSACS'99*, volume 1578 of *LNCS*, pages 14–30, 1999.

[17] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV: International Conference on Computer Aided Verification*, 2001. (to appear).

[18] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, 1995.

[19] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY'97: Verification of Infinite-state systems*, 1997.

[20] Li Gong. *Java^{TM} Security Architecture (JDK 1.2)*. Sun Microsystems, 1998.

[21] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Verlag, 1997.

[22] K. Havelund and J. U. Skakkebæk. Applying model checking in Java verification. In *Proceedings of the 6th SPIN Workshop*, pages 216–231, Toulouse, France, 1999.

[23] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[24] S. Hudson. Cup LALR parser generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[25] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of java multithreading applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, Paris, France, 1998.

[26] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties, 1999.

[27] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[28] G. Klein. JFlex - The fast lexical scanner generator for Java. `http://www.jflex.de/`.

[29] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science (TCS)*, 27:333–354, 1983.

[30] R. Mayr. *Decidability and Complexity of Model Checking Problem for Infinite-State Systems*. PhD thesis, Technische Universität München, 1998.

[31] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[32] F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[33] D. M. R. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, volume 104 of *LNCS*, pages 167–183, 1981.

[34] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Århus University, 1981.

[35] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[36] D. Polanský. Implementation of the modelchecker for pushdown systems and alternation-free mu-calculus. Master's thesis, FI MU Brno, 2000.

[37] AT&T Labs Research. Graphviz - open source graph drawing software. `http://www.research.att.com/sw/tools/graphviz`.

[38] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[39] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35.10 of *ACM Sigplan Notices*, pages 264–280. ACM Press, 2000.

[40] R. Vallée-Rai, Phong Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON '99*, 1999.

[41] R. van Glabbeek. The linear time – branching time spectrum. Technical Report TUM-I9025, Technische Universität München, July 1990.

[42] W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the Fourth ACM Workshop on Formal Methods in Software Practice*, Portland, Oregon, August 2000.

[43] I. Walukiewicz. Pushdown processes: games and model checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV'96)*, volume 1256 of *Lecture Notes in Computer Science*, 1996.