Lehrstuhl für Informatik VII
der Technischen Universität München

Model-Checking Pushdown Systems

Stefan Schwoon

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. T. Nipkow, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Dr.h.c. W. Brauer 2. Prof. Dr. J. Esparza, Universität Edinburgh

Die Dissertation wurde am 27.06.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.11.2002 angenommen.

Abstract

The thesis investigates an approach to automated software verification based on pushdown systems. Pushdown systems are, roughly speaking, transition systems whose states include a stack of unbounded length; there is a natural correspondence between them and the execution sequences of programs with (possibly recursive) subroutines. The thesis examines model-checking problems for pushdown systems, improving previously known algorithms in terms of both asymptotic complexity and practical usability.

The improved algorithms are used in a tool called Moped. The tool acts as a model-checker for linear-time logic (LTL) on pushdown systems. Two different symbolic techniques are combined to make the model-checking feasible: automata-based techniques are used to handle infinities raised by a program's control, and Binary Decision Diagrams (BDDs) to combat the state explosion raised by its data.

It is shown how the resulting system can be used to verify properties of algorithms with recursive procedures by means of several examples. The checker has also been used on automatically derived abstractions of some large C programs. Moreover, the thesis investigates several optimizations which served to improve the efficiency of the checker.

Acknowledgements

This thesis would not have been possible without the guidance, generosity, and goodwill of many people. I feel grateful and indebted to have received all their help.

To a large part, this applies to my supervisor Prof. Javier Esparza, who gave me the opportunity to conduct research in his group and introduced me to a topic that suited my interests. I also have to thank Prof. Wilfried Brauer in his twofold capacity as referee and as speaker of the graduate college "Co-operation and resource management in distributed systems". The Deutsche Forschungsgemeinschaft (DFG), which funded the graduate college, gave me indispensable financial support, and both the Technical University of Munich and the University of Edinburgh, where I studied, provided organizational and additional financial help.

I enjoyed my time at both institutions mainly because of my colleagues and the great working atmosphere in our group. I thank them all, especially Claus Schröter and Peter Rossmanith, who always had an open ear for discussions about data structures, algorithms, and many other things. I also want to acknowledge David Hansel, who wrote the original implementation of the tool that eventually grew into Moped.

Moreover, parts of the thesis are due to the direct influence of members of the international research community. Ahmed Bouajjani helped to develop the theory on which this work is built and continued to provide valuable ideas throughout. Antonín Kučera inspired the work on regular valuations; Chapter 6 directly results from joint work between him and me. Tom Ball and Sriram Rajamani not only invited me to visit them at Microsoft Research, but also provided me with many examples on which I could try my model-checker. This opportunity has been a most valuable and motivating experience for me.

Most importantly, however, my parents enabled me to study in the first place, and their love and encouragement never failed to support me throughout the years. Thank you for everything.

Contents

Chapter 1

Introduction

Research on formal verification concerns the theory and practice of methods to prove that computer systems meet critical design requirements. It has implications on everyday life in today's society where computers and electronic equipment in general assume more and more functions. As our dependence on these systems grows, so does the need to ensure that the hardware and software components they consist of deserve the trust that we put into them.

1.1 Verification vs testing

The most basic form of verification is testing. In testing, the actual behaviour of the system under consideration (be it a piece of hardware or a computer program) is observed on a set of inputs and matched against the expected behaviour. In hardware verification, where fabrication runs are costly, the testing of actual samples is often preceded by a simulation stage where the behaviour of the system is modelled by a software program.

Simulation and testing are essential parts of a development cycle, but they suffer from fundamental shortcomings, the main one being that they are incomplete: If the set of possible inputs becomes very large or even infinite, it becomes impossible to confirm that the system behaves correctly in all possible circumstances, and so errors may not be noticed. Strictly speaking, testing may not even guarantee that subsequent runs on the same input yield the same result. Moreover, the tester is required to determine what the correct behaviour of a program actually is; this becomes increasingly difficult as the system grows in complexity. The approach does not translate well to reactive systems, where the behaviour consists of continual interaction with its environment rather than producing an output and terminating. In fact, testing cannot distinguish a program that is not terminating (because of an error) from one whose computation merely takes very long. Lastly, testing can merely assert the presence of an error, but not provide an explanation of how things went wrong and why.

Formal verification tries to address some of these problems. Instead of merely observing the systems behaviour, it provides foundations to reason about correctness of a system on a mathematical basis. In order to achieve this, the verification problems need to be formalised; what is required is a precise mathematical model of the behaviour of the system and a precise definition of its intended behaviour. For the former, a machine is usually modelled as a system whose state evolves over time; the model then consists of a specification of the state space and how the system can traverse it. For the latter, temporal logic has been shown to be useful to express the behaviour of reactive systems as well as properties like program termination, the existence of deadlocks etc.

The model-checking approach to formal verification consists of taking a mathematical model of the system under consideration and checking the validity of a temporal logic formula within the model. A number of interesting model-checking problems has been shown to be decidable and can hence be automated and solved by computers. Many model-checking methods not only determine the validity of a temporal logic formula but, in the case of failures, also find execution sequences that violate it; an engineer can use such a sequence to find and repair an error in the system.

Formal verification is the subject of large and varied research efforts. In the model-checking community, researchers have investigated many different mathematical models and temporal logics. The diversity stems from the fact that formal verification is fundamentally hard; there is no single method to accommodate all interesting problems. The primary problem faced by all methods is popularly known as state explosion, meaning that the state space of the system under consideration typically grows exponentially with the amount of memory used by it (i.e. registers, or program variables, instruction pointers etc.), which limits the practical applicability of verification methods. The methods used to counter the state explosion problem usually fall into three categories: reduction, abstraction, and compression.

Reduction and abstraction both replace the original complete model by a simplified one with a smaller state space on which the property of interest is then checked. Reduction can be characterized as cutting away states and execution sequences without affecting the validity of the property in question. For instance, if a system can execute one or more steps in any order, but the order in which the steps happen has no influence on whether the property holds or not, then nothing is gained by exploring all possible orders, and one can reduce the system to one where some arbitrary order is fixed.

Abstraction techniques partition the state space. The abstract model has one abstract state for each partition and can move from one abstract state to another if and only if their respective partitions contain states which are connected in the original system. Thus, information is lost and the abstract model is an overapproximation of the original model. For instance, the abstract model might use less variables than the original model and employ non-determinism whenever the value of a removed variable is queried in order to simulate all possible outcomes. This way, errors that do exist in the original model are preserved, but new 'spurious' errors may be added. If such a spurious error is detected the abstraction has to be refined, possibly repeatedly.

Compression concerns the use of efficient data structures to succinctly represent large sets of states in the hopes that 'well-behaved' verification problems will be amenable to such representations. These techniques are also often called symbolic (as opposed to explicit). Well-known symbolic representations include Binary Decision Diagrams, finite automata, polyhedra, and others.

1.2 Hardware vs software verification

Traditionally, the verification community distinguishes between two fields of application: hardware, and software. While the verification of hardware has seen significant progress and has advanced into large-scale industrial use, the same cannot be said of software. The reasons for this appear to be partly technical (availability and usability of methods), and partly economical.

In the hardware industry, the need for verification of digital designs is driven by the high cost of production. A chip whose design does not meet its specification is essentially worthless; its production cost is lost, and parts that have already been shipped to customers cannot easily be replaced (and only with a loss of reputation). Large chip manufacturers are therefore quite prepared to spend more effort in the pre-production phase to ensure the validity of their designs, and formal verification techniques have become ubiquitous in this area. From a model-checking point of view, hardware can be modelled by finite state systems; thus, model-checking problems remain decidable and can be automated. In practice, complex processor designs with several hundreds of variables have been successfully verified.

Software verification, on the other hand, is a rather harder problem – not only in practice where programs can be vastly complex, but also on the theoretical side: Since a computer program has the same expressiveness as a Turing machine, even very simple verification problems are generally undecidable. It is probably in subconscious recognition of this that the presence of bugs in computer software (at least as far as non-safety-critical software is concerned) appears to be accepted as a necessary evil by software companies and a many customers alike. Hence, the software industry is currently using formal verification methods on a much smaller scale than the hardware industry. Another contributing factor is that software can be more easily replaced than hardware, so in economic reality the extra cost of thorough verification and longer time-to-market often does not pay off.

For all the aforementioned reasons, general *and* easily usable solutions for verification of software are an unrealistic goal. Research therefore concentrates on identifying subclasses of programs which are suitably expressive but not Turing-powerful and still allow automated verification, and on applying them to 'essential' aspects of software, e.g. cryptographic protocols or communication protocols.

1.3 Contribution of the thesis

This thesis investigates an approach to automated software verification. The abstract formalism employed to describe programs is known as *pushdown systems*. Pushdown systems are, roughly speaking, transition systems whose states include a stack of unbounded length; hence, they are strictly more expressive than finite state systems. One can argue that pushdown systems are a natural model for sequential programs with procedures where there is no restriction on the call hierarchy among the procedures. Arbitrary recursion is allowable since the stack can keep track of active procedure calls. The model has two main restrictions: it does not handle parallelism, and it can only model finite data domains.

The pushdown model and its relation to programs with procedures is explained in greater detail in Chapter 2. This section also discusses other preliminaries, such as symbolic techniques to efficiently represent the model and its state space, and temporal logics.

Chapter 3 then examines model-checking problems for pushdown systems, starting with reachability and going on to linear-time logic (LTL). It extends the theory presented in [9, 22], which served as a starting point. The previously known algorithms are improved in terms of both asymptotic complexity and practical usability with suggestions for data structures and implementation details to speed them up.

The improved algorithms are used in a tool called Moped. The tool acts as a model-checker for linear-time logic on pushdown systems and combines two different symbolic techniques to make the model-checking feasible: automata-based techniques are used to handle infinities raised by a program's control, and Binary Decision Diagrams (BDDs) to counter the state explosion raised by its data. Moped has been made available to the public [37] and is described in greater detail in Chapter 4. As a usability test, the checker has been applied to automatically derived abstractions of large (10,000 lines of code) device drivers written in C. In these tests, Moped managed to perform better than Bebop, a similar model-checker that was developed concurrently [3].

Efforts have been made to ensure the efficiency of the checker, which is influenced by a variety of design choices. Chapter 5 lists some of these choices and evaluates their impact by means of further experiments.

Chapter 6 discusses an extension of the model-checking problems from Chapter 3 in which the predicates of LTL formulas may be given by regular languages. This extension provides a unified framework for problems from different areas (e.g. data-flow analysis, analysis of systems with checkpoints), and can also be used to extend the model-checker to the more powerful logic CTL*.

1.4 Related research

Pushdown systems have already been investigated by the verification community in the past. Model-checking algorithms for various temporal logics have been proposed, e.g. by Walukiewicz [42] and Burkart and Steffen [12] who have shown decidability of the full-modal mu-calculus for pushdown systems. While the model-checking problem for branching-time logics is computationally intractable (even for the relatively small fragment EF), linear-time logic can be solved in polynomial time (for a fixed formula). Finkel et al [22] and Bouajjani et al [9] first proposed such algorithms. Esparza and Podelski showed that the reachability problem remains polynomial even for parallel interprocedural flow graphs [21] which are more expressive than pushdown systems. The various complexity and decidability results are discussed in greater detail in later sections.

On the implementation side, Polanský [35] has implemented a model-checker for the alternation-free mu-calculus based on the algorithm in [9]. Moreover, Obdržálek has applied the Moped checker to the verification of Java programs [33].

Jensen, Le Métayer and Thorn [26] introduced a model that could be described as pushdown systems with checkpoints. This model is motivated by languages like newer versions of Java, where programs can perform local security checks in which it is verified that the methods on the call stack have the appropriate permissions to perform an action. In [26] this was used to prove the validity of global security properties as well as to detect (and remove) redundant security checks. In Chapter 6 this thread is taken up again; the methods presented here provide an efficient framework for this application and are somewhat more general than those of [26].

A number of other researchers have investigated a model of computation related to pushdown systems called recursive (or hierarchical) state machines (RSMs). In fact, pushdown systems and RSMs possess the same expressiveness and succinctness but could be said to operate from different perspectives: RSMs explicitly model a procedural setting whereas pushdown systems take on a more 'machine-oriented' view. Recently, two papers by Benedikt, Godefroid, Reps [6] and Alur, Etessami, Yannakakis [1] proposed efficient solutions for reachability problems and linear-time logic on RSMs. Their algorithms provide a slight asymptotic improvement for model-checking RSMs as compared to translating an RSM into a pushdown system and applying the methods from [18]. However, no implementation for these algorithms exists. A more thorough examination of the relation between the two models and a comparison of the algorithms put forward for them is provided in Section 3.1.5.

Remarkable efforts towards a verification framework for large-scale C programs have been made by Ball and Rajamani on the basis of Boolean programs, which could be dubbed as hierarchical state machines equipped with Boolean variables [4]. Their SLAM toolkit implements a cyclic process based on abstraction and iterative refinement. Each iteration of the cycle consists of three steps. In the first step, a C program is fully automatically abstracted into a Boolean Program. In the second step, the reachability checker Bebop is used to search for violations of the desired safety property. Because the abstracted Boolean Program overapproximates the possible execution sequences of the C program, resulting error traces are then checked for feasibility using a third tool; if the error trace is spurious, the cycle starts again with a more refined abstraction. This iteration takes place until either no error trace is found anymore, or a real error is found. The process is not guaranteed to terminate but has shown good results in practice. Ball and Rajamani are using SLAM to verify that device drivers used in Windows observe a protocol imposed by the operating system. Moped has been equipped with an interface to Boolean Program and the SLAM kit and could be used in place of Bebop within the verification cycle.

Chapter 2

Preliminaries

A model-checking framework consists of three parts; a mathematical model for the behaviour of the systems which are to be verified, a logic for defining properties of these systems, and a method that checks whether a given system satisfies a given property. This section introduces the first two items of the framework laid out in this thesis.

2.1 Basic definitions

In the following, we assume the usual notations for alphabets, concatenation, finite and infinite words:

Definition 2.1 *Let S be a set (or* alphabet*). A (finite)* word *w over S is a finite sequence of elements of S. The empty word is denoted by ε, non-empty sequences by the concatenation of their elements. The* length *of w is denoted by $|w|$, i.e. if $w = s_1 \dots s_n$, then $|w| = n$. Moreover, $w^R := s_n \dots s_1$ denotes the reverse of w.*

S^* is the set of all words over S, and $S^+ := S^* \setminus \{\varepsilon\}$ denotes the set of all non-empty words. S^ω is the set of all infinite sequences (or ω-words) of elements of S. A language *over S is some set of finite or infinite words over S.*

Within this thesis, finite words will usually be referred to by lowercase letters near the end of the alphabet such as u, v, w, possibly with indices. Infinite sequences will usually be denoted by lowercase Greek letters near the end of the alphabet such as σ or ω.

An infinite sequence σ can also be seen as a function from $\mathbb{N}_0$ to S; hence $\sigma = \sigma(0)\sigma(1)\ldots$. Given an ω-word σ we denote by σ^i (for any $i \geq 0$) the i-th suffix of σ, i.e. the ω-word $\sigma(i)\sigma(i+1)\ldots$. The set $\mathit{Inf}(\sigma)$ contains exactly those elements of S which occur infinitely often in σ.

2.2 Transition systems

In order to reason about the behaviour of a system it is convenient to think of it as an entity which can assume different states. A mathematical definition of possible behaviours would then describe the set of possible states (the *state space*) and how the system can evolve from one state into the other (the *transition relation* between states). In the following, we assume that time is discrete; in other words, a behaviour will consist of an enumerable number of states. While this is at odds with the natural notion that time is continuous, it is a often a good enough approximation, in particular for digital systems whose components operate according to a discrete tact. The most general model for such descriptions are *labelled transition systems.*

Definition 2.2 *A labelled transition system is a quadruple $\mathcal{T} = (S, A, \rightarrow, r)$ where S is a set of* states, *A is a set of* actions, *$\rightarrow \subseteq S \times A \times S$ is a* transition relation, *and r is a distinguished state called* root. *The transition relation $\rightarrow$ denotes possible state changes; if $(s, a, s') \in \rightarrow$ we say that the system can move from state s to s' by performing action a. As a more compact notation, we usually write $s \xrightarrow{a} s'$. If $s \xrightarrow{a} s'$ for some a, then s is an* immediate predecessor *of s', and s' is an* immediate successor *of s.*

In order to reason about the behaviour of such a system, we need to introduce several reachability relations:

Definition 2.3 *A* path *of $\mathcal{T}$ is a sequence of states $s_0 \ldots s_n$ (for some $n \geq 0$) such that the system can 'evolve' from s_0 to s_n, i.e. there are transitions $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We say that such a path has length n and is labelled by the word $a_1 \ldots a_n \in A^*$.*

If there is a path of length n from s to s' labelled by w, then we write $s \xrightarrow[n]{w} s'$. We may also write $s \xrightarrow{w}^+ s'$ to denote that $s \xrightarrow[n]{w} s'$ holds for some $n \geq 1$. Moreover, $s \xrightarrow{w}^* s'$ holds if and only if $s \xrightarrow{w}^+ s'$ or $w = \varepsilon$ and $s = s'$.

If $s \xrightarrow{w}^* s'$ for some $w \in A^*$, then s' is said to be reachable from *s. If s' is reachable from the root r, then s' is* reachable. *Given a set $S' \subseteq S$, the* *set* $pre^*(S') = \{ s \mid \exists s' \in S', w \in A^* \colon s \xrightarrow{w}{}^* s' \}$ *contains the* predecessors *of* S'*, and the set* $post^*(S') = \{ s \mid \exists s' \in S', w \in A^* \colon s' \xrightarrow{w}{}^* s \}$ *contains the* successors *of* S'*.*

A run *of* $\mathcal{T}$ *is an infinite sequence of states* $\sigma = s_0 s_1 \ldots$ *where for all* $i \geq 0$ *we have* $s_i \xrightarrow{a_{i+1}} s_{i+1}$*. Such a run is said to by labelled by the* ω*-word* $\omega = a_1 a_2 \ldots$*.*

A labelled transition system can be identified with a directed graph whose nodes are elements of S and which has an edge from node s to node s' labelled by a if $s \xrightarrow{a} s'$. We borrow some definitions from graph theory:

Definition 2.4 *Let* $\mathcal{T} = (S, A, \rightarrow, r)$ *be a labelled transition system. A set of states* $S' \subseteq S$ *is a* strongly connected component *(SCC) of* $\mathcal{T}$ *if and only if every pair of states in* S' *is reachable from each other and* S' *is maximal, i.e. there is no set* $S'' \supset S'$ *which has this property. An SCC is* trivial *if it consists of a single state which has no transition to itself. The* diameter *of* $\mathcal{T}$ *is the length of the longest path between any two states in* S*.*

In the definition above, a labelled transition system defines both the possible evolutions of a system through time and the actions taken during each particular evolution. Depending on our needs, we may wish to emphasize one aspect over the other. If, for instance, we are interested only in the possible behaviours but not in the actions which are carried out, we can use the simpler notion of an (unlabelled) transition system $(S, \rightarrow, r)$. All the relevant notions from this subsection are defined analogously for unlabelled transition systems; we drop the labels from the reachability relations and write $s \rightarrow s'$, $s \rightarrow^* s'$, and $s \xrightarrow[n]{} s'$, respectively.

Alternatively, we can focus on the sequences of actions that are possible in the system. We then define the semantics of a transition system to be the set of labels of paths or runs which adhere to some *acceptance condition*. Thus, a transition system $(S, A, \rightarrow, r)$ can be used as a symbolic representation of a language over the set of actions A.

Having established the notations above, we introduce some well-known classes of transition systems relevant to the material that follows later.

2.2.1 Finite automata

A finite automaton is a language acceptor. Informally speaking, the languages represented by finite automata are those that can be recognized by a machine or program which reads its input only once, stores only one element of the input at a time, and requires a constant amount of memory.

Definition 2.5 *A finite automaton is a quintuple $\mathcal{M} = (Q, \Gamma, \rightarrow, q_0, F)$ such that Q is a finite set of states, Γ is a finite set called the* input alphabet*, $\rightarrow \subseteq (Q \times \Gamma \times Q)$ is a set of* transitions*, $q_0 \in Q$ is the* initial state*, and $F \subseteq Q$ is the set of* final states*. The language $L(\mathcal{M})$ accepted by $\mathcal{M}$ is the set of all finite words w such that $q_0 \xrightarrow{w}{}^{*} q'$ holds for some final state $q' \in F$ in the labelled transition system $(Q, \Gamma, \rightarrow, q_0)$. A language is called* regular *if and only if there is a finite automaton which accepts it.*

2.2.2 Büchi automata

A Büchi automaton is similar to a finite automaton with the difference that its acceptance condition works on ω-words.

Definition 2.6 *A Büchi automaton $\mathcal{B}$ is a quintuple $(Q, \Gamma, \rightarrow, q_0, F)$ where Q is a finite set of states, Γ is a finite* input alphabet*, $\rightarrow \subseteq (Q \times \Gamma \times Q)$ is a set of* transitions*, and $q_0 \in Q$ is the* initial state*. The language $L(\mathcal{B})$ accepted by $\mathcal{B}$ is the set of all infinite sequences ω such that the labelled transition system $(Q, \Gamma, \rightarrow, q_0)$ has a run σ labelled by ω and starting at q_0 such that $Inf(\sigma) \cap F \neq \emptyset$, i.e. σ visits accepting states from F infinitely often. A language is called* ω-regular *if and only if there is a Büchi automaton which accepts it.*

2.2.3 Pushdown systems

A pushdown system is a transition system equipped with a finite set of *control locations* and a *stack*. The stack contains a word over some finite *stack alphabet*; its length is unbounded. Hence, a pushdown system may have infinitely many reachable states.

Definition 2.7 *A pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ is a quadruple where P contains the control locations and Γ is the stack alphabet. A* configuration *of $\mathcal{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted by $Conf(\mathcal{P})$. With $\mathcal{P}$ we associate the unique transition system $\mathcal{T}_{\mathcal{P}} = (Conf(\mathcal{P}), \Rightarrow_{\mathcal{P}}, c_0)$ whose initial state or configuration is c_0.*

Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$; if $((p, \gamma), (p', w)) \in \Delta$, we also write $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$. The transition relation of $\mathcal{T}_{\mathcal{P}}$ is determined by the set of rules Δ *as follows:*

$$\textit{If } \langle p,\gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p',w\rangle, \textit{ then } \langle p,\gamma w'\rangle \Rightarrow_{\mathcal{P}} \langle p',ww'\rangle \textit{ for all } w' \in \Gamma^*.$$

The head *of a configuration $\langle p,\gamma w\rangle$ is the pair $\langle p,\gamma\rangle$. Similarly, the head of a rule $\langle p,\gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p',w\rangle$ is the pair $\langle p,\gamma\rangle$.*

In other words, a transition in $\mathcal{T}_{\mathcal{P}}$ may change the control location and replace the 'topmost' (i.e. leftmost) stack symbol by a string of other symbols including the empty string (effectively removing the topmost symbol). Every step depends only on the control location and the topmost element of the stack. The rest of the stack is unchanged and has no influence on the possible actions. In the following, if the pushdown system in question is understood, we drop the index $\mathcal{P}$ from the relations $\Rightarrow_{\mathcal{P}}$ and $\hookrightarrow_{\mathcal{P}}$.

The focus of this thesis is on the analysis of pushdown systems and its use in verification of software. The next section illustrates this application.

2.3 A model of sequential programs

Pushdown systems find a natural application in the analysis of sequential programs, e.g. programs written in languages such as C or Java. These are typically composed of a number of procedures. Procedures may interact by calling each other, passing argument values to the callee or returning values to the caller. An execution of a function may involve calls to other functions or even recursively to itself. When a function terminates, execution resumes at its caller. Execution starts in a distinguished function *main* which cannot be called recursively. As we shall see, many aspects of such programs have a natural counterpart in the behaviour of pushdown systems. In the following, we discuss informally how to derive a pushdown system from such a program.

2.3.1 Control flow

If we are interested in modelling only the control flow of a program, we proceed in two steps.

In the first step, we represent the program by a system of *flow graphs*, one for each procedure. The nodes of a flow graph correspond to control points in its associated procedure, and its edges are annotated with statements, e.g. assignments or calls to other procedures. Non-deterministic control flow is allowed and may for instance result from abstraction. Each flow graph has a distinguished start node.

```
void m() {                          void s() {
    if (?) {                            if (?) return;
        s(); right();                   up(); m(); down();
        if (?) m();                 }
    } else {
        up(); m(); down();          main() {
    }                                 s();
}                                   }
```

Figure 2.1: An example program.

As an example, consider the program in Figure 2.1. The program, presented in C-like pseudo-code, controls a plotter, creating random bar graphs via the commands *up*, *right*, and *down*. Figure 2.2 displays the set of flow graphs created from the program. (We represent *up*, *right*, and *down* by flow graphs with just a return action. Note that the behaviour of this example could not be modelled by a finite state system since there is no bound on the depth of the recursion.

In the second step, we apply a straightforward transformation to obtain a pushdown system. Let N be the union of the node sets of all flow graphs (assuming that the node sets of different procedures are disjoint). Then we construct a pushdown system $(\{\cdot\}, N, \Delta, \langle \cdot, main_0\rangle)$ with a single control location. The stack alphabet consists of the flow graph nodes where $main_0$ should be the start node of the procedure *main*. The rewrite rules are as follows:

- $\langle \cdot, n\rangle \hookrightarrow \langle \cdot, n'\rangle$ if control passes from n to n' without a procedure call (for instance, through an assignment).

- $\langle \cdot, n\rangle \hookrightarrow \langle \cdot, f_0 n'\rangle$ if an edge between points n and n' contains a call to procedure f, assuming that f_0 is f's entry point; n' can be seen as the return address of that call.

- $\langle \cdot, n\rangle \hookrightarrow \langle \cdot, \varepsilon\rangle$ if an edge leaving n contains a return statement.

Thus, a configuration $\langle \cdot, nw\rangle$ represents the fact that execution is currently at node n whereas w represents the return addresses of the calling procedures. Figure 2.3 shows the rewrite rules derived from the example. Notice that the resulting pushdown system does not have any notion of procedures;

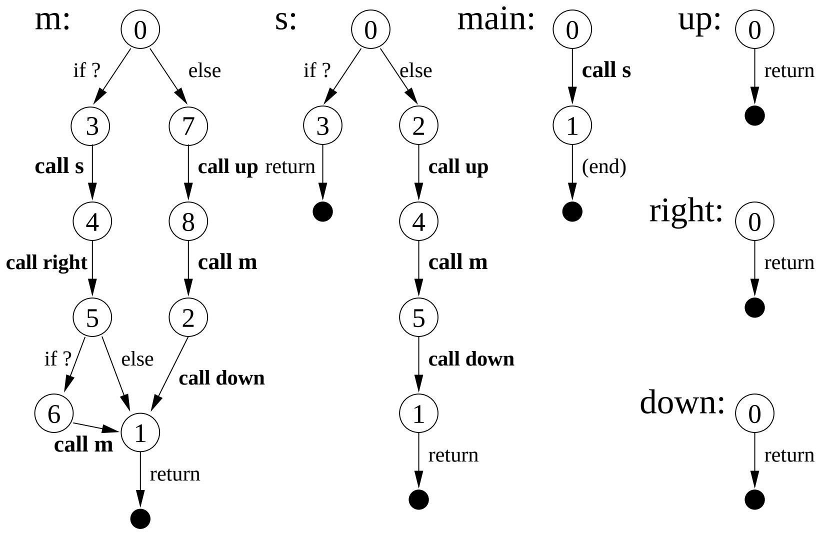

Figure 2.2: Flow graph of the program in Figure 2.1.

it uses the stack merely to record the effect of call and return statements but does not require the program to have any particular structure.

2.3.2 Data

The construction in the previous subsection produces pushdown systems which are overapproximations of their underlying programs. Since we remove all variables, we cannot decide when a conditional jump has to be made; therefore we allow both possibilities non-deterministically. Thus, the possible runs of the derived pushdown system are a superset of the runs which are possible in the underlying program. For many applications this approximation is too rough, i.e. complete abstraction of variable contents is too restrictive.

Procedural programming languages typically distinguish between global variables, which are accessible to all procedures, and local variables. Each procedure may declare a set of local variables which are accessible only to itself. More precisely, whenever the procedure is invoked, new instances of its local variables are created; these instances are accessible only to that particular invocation of the function, and they are destroyed upon termination of the procedure.

These mechanisms can be simulated quite faithfully by the behaviour of a

$$
\begin{array}{ll}
\langle \cdot, m_0 \rangle \hookrightarrow \langle \cdot, m_3 \rangle & \langle \cdot, m_0 \rangle \hookrightarrow \langle \cdot, m_7 \rangle \\
\langle \cdot, m_3 \rangle \hookrightarrow \langle \cdot, s_0\, m_4 \rangle & \langle \cdot, m_4 \rangle \hookrightarrow \langle \cdot, right_0\, m_5 \rangle \\
\langle \cdot, m_5 \rangle \hookrightarrow \langle \cdot, m_1 \rangle & \langle \cdot, m_5 \rangle \hookrightarrow \langle \cdot, m_6 \rangle \\
\langle \cdot, m_6 \rangle \hookrightarrow \langle \cdot, m_0\, m_1 \rangle & \langle \cdot, m_7 \rangle \hookrightarrow \langle \cdot, up_0\, m_8 \rangle \\
\langle \cdot, m_8 \rangle \hookrightarrow \langle \cdot, m_0\, m_2 \rangle & \langle \cdot, m_2 \rangle \hookrightarrow \langle \cdot, down_0\, m_1 \rangle \\
\langle \cdot, m_1 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle & \\[1ex]
\langle \cdot, s_0 \rangle \hookrightarrow \langle \cdot, s_2 \rangle & \langle \cdot, s_0 \rangle \hookrightarrow \langle \cdot, s_3 \rangle \\
\langle \cdot, s_2 \rangle \hookrightarrow \langle \cdot, up_0\, s_4 \rangle & \langle \cdot, s_3 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle \\
\langle \cdot, s_4 \rangle \hookrightarrow \langle \cdot, m_0\, s_5 \rangle & \langle \cdot, s_5 \rangle \hookrightarrow \langle \cdot, down_0\, s_1 \rangle \\
\langle \cdot, s_1 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle & \\[1ex]
\langle \cdot, main_0 \rangle \hookrightarrow \langle \cdot, s_0\, main_1 \rangle & \langle \cdot, main_1 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle \\[1ex]
\langle \cdot, up_0 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle & \langle \cdot, down_0 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle \\
\langle \cdot, right_0 \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle &
\end{array}
$$

Figure 2.3: Pushdown system associated with the flow graphs in Figure 2.2.

pushdown system if the global variables are encoded into the control locations and the local variables into its stack alphabet. The main limitation of this approach comes from the requirement that both the control locations and the stack alphabet must be finite sets. Essentially this means that only variables of finite data types can be modelled, e.g. boolean variables or integers from a finite fragment. (Strictly speaking, since the stack can be arbitrarily long, a configuration can still encode an unbounded amount of data, but the pushdown system can access it only in a restricted fashion.) Notice that allowing an infinite set of control locations would make the model Turing-powerful and render all the interesting analysis problems undecidable. By employing abstraction, we can still use pushdown systems to argue about programs with more complex data types *provided* that the properties of those data structures relevant to the application can be expressed by a finite number of boolean predicates (see also Section 4.2.3).

With the aforementioned encoding, the control location becomes a tuple g which represents the values of the global variables, whereas the stack alphabet contains pairs (n, l) where n is a control point (as above) and l is a tuple carrying the local variables of the procedure that n belongs to, but does not store the local variables of any other procedure. Thus, the precise meaning of l depends on n. This encoding preserves characteristic properties of global and local variables. For instance, each step in the pushdown system depends only on the current control location and the topmost stack symbol; this corresponds to the property that at each point in time a program can only access and/or modify the global variables and the local variables of the currently active procedure. Moreover, since local variables are stored on the stack, the pushdown system can simulate the instantiation of new copies when recursive calls are made.

Let us revisit the translation process from the previous subsection. Assume that G is the domain of the global variables (i.e., G could be the cross product of domains of the individual variables). Moreover, if we have m procedures and L_i, $1 \leq i \leq m$, are the cross-products of the domains of their respective local variables, then let L be a set with $|L| \geq |L_i|$ for all $1 \leq i \leq m$ such that the values from any of the L_i can be encoded within L. (As we shall see later, this encoding has an impact on the asymptotic complexity of the algorithms we devise to analyse the systems.) We then construct a pushdown system $(G, N \times L, \Delta', \langle g_0, (main_0, l_0)\rangle)$ where g_0 and l_0 are the initial values of the global and local variables of *main*, respectively. Each program statement is translated to a set of pushdown rules as follows:

Constraints If a flow graph edge from node n_1 to node n_2 represents an assigment or another 'simple' statement (i.e. anything other than a function call or a return statement), we translate it to set of rules of the form

$$\langle g, (n_1, l)\rangle \hookrightarrow \langle g', (n_2, l')\rangle.$$

where g and g' (l and l') are the values of the global (local) variables before and after the assigment; for each pair (g, l), the new values (g', l') describe a possible effect of the statement. In the case of an assignment, for instance, g and l would be the same as g' and l' except for the entries of the variable(s) on the left-hand side of the assignment. Alternatively, if the edge represents an 'if' or 'while' statement, we could restrict the values of g and l to the set allowed by its branching condition.

Procedure calls If a flow graph edge from node n_1 to node n_2 represents a procedure call to a function m (with start node m_0), it will be translated into a set of rules with a right-hand side of length two according to the following scheme:

$$\langle g, (n_1, l)\rangle \hookrightarrow \langle g, (m_0, l')\,(n_2, l)\rangle$$

```
bool l; /* global variable */

void lock() {
     if (l) ERROR;
     locked := 1;
     ··· /* acquire a lock */
}

void unlock() {
     if (!l) ERROR;
     ··· /* release the lock */
     locked := 0;
}
```

```
bool g (bool x) {
     return !x;
}

void main() {
     bool a,b;
     l,a := 0,0;
     ...
     lock();
     b := g(a);
     unlock();
     ...
}
```

Figure 2.4: Another example with boolean variables.

Here, l' denotes initial values of the local variables of m as well as any arguments that may be passed to the function; the local variables of the calling function are stored on the stack together with the return address n_2.

Return statements A return statement has an empty right-hand side:

$$\langle g, (n, l)\rangle \hookrightarrow \langle g', \varepsilon\rangle$$

If the procedure has return values, these have to be encoded in the global variables, possibly by introducing temporary new variables.

Example Consider the pseudo-code program shown in Figure 2.4. The program has a global boolean variable `l` which is supposed to determine whether the program has exclusive access (a lock) on some file or device. The functions `lock` and `unlock` acquire and release the lock; it is an error to acquire a lock twice without releasing it in between or vice versa. The function `g` takes a boolean argument and returns a boolean value. The function `main` has two local variables `a` and `b`. The values 'true' and 'false' are represented by `T` and `F`, respectively.

The corresponding flow graph is shown in Figure 2.5, the pushdown rules in Figure 2.6. The control locations contain the value of the global variable `l` and, when needed, the return value of function `g` (see rules 10 and 15).

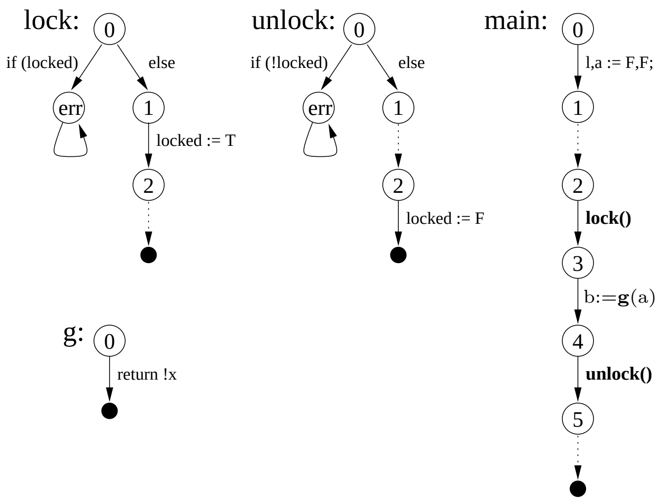

Figure 2.5: Flow graph of the program in Figure 2.4.

The stack alphabet consists of control points and the local variables of their associated procedures. Rules 3, 9, and 11 are translations of assignments; rules 1 and 2, and 6 and 7, are examples for 'if' statements. Procedure calls are made in rules 13, 14 and 16. Rule 14 shows how to pass an argument to g. In order to evaluate the return value of g, control is first transferred to an intermediate control location $main_{3'}$ after the call. Rule 15 then shows the evaluation of the return value from a procedure.

In general, the translation yields a pushdown system whose size is far larger than the underlying program. This is because in each statement one rule is needed for every admissible combination of variable values; it is easy to see that the number of resulting rules is exponential in the number of variables. Therefore it may quickly become impractical to represent the system by an explicit enumeration of the rules. What appears to be needed are data structures for storing rules in a similar notation to that used in Figure 2.6: Instead of enumerating the rules, we can represent sets of them with the control flow information accompanied by a compact representation of the admissible valuations, effectively separating the representations of control and data. The following paragraph concretizes this idea.

Symbolic pushdown systems Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system where the control locations are of the form $P = P_0 \times G$ and the stack

$$
\begin{array}{lll}
(1) & \langle(\mathtt{T}),(lock_0)\rangle \hookrightarrow \langle(\mathtt{T}),(err)\rangle & \\
(2) & \langle(\mathtt{F}),(lock_0)\rangle \hookrightarrow \langle(\mathtt{F}),(lock_1)\rangle & \\
(3) & \langle(l),(lock_1)\rangle \hookrightarrow \langle(\mathtt{T}),(lock_2)\rangle & l \in \{\mathtt{T},\mathtt{F}\} \\
(4) & \langle(l),(lock_2)\rangle \hookrightarrow \langle(l),\varepsilon\rangle & l \in \{\mathtt{T},\mathtt{F}\} \\
(5) & \langle(l),(err)\rangle \hookrightarrow \langle(l),(err)\rangle & l \in \{\mathtt{T},\mathtt{F}\} \\
(6) & \langle(\mathtt{F}),(unlock_0)\rangle \hookrightarrow \langle(\mathtt{F}),(err)\rangle & \\
(7) & \langle(\mathtt{T}),(unlock_0)\rangle \hookrightarrow \langle(\mathtt{T}),(unlock_1)\rangle & \\
(8) & \langle(l),(unlock_1)\rangle \hookrightarrow \langle(l),(unlock_2)\rangle & l \in \{\mathtt{T},\mathtt{F}\} \\
(9) & \langle(l),(unlock_2)\rangle \hookrightarrow \langle(\mathtt{F}),\varepsilon\rangle & l \in \{\mathtt{T},\mathtt{F}\} \\
(10) & \langle(l),(g_0,x)\rangle \hookrightarrow \langle(l,\neg x),\varepsilon\rangle & l,x \in \{\mathtt{T},\mathtt{F}\} \\
(11) & \langle(l),(main_0,a,b)\rangle \hookrightarrow \langle(\mathtt{F}),(main_1,\mathtt{F},b)\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\} \\
(12) & \langle(l),(main_1,a,b)\rangle \hookrightarrow \langle(l),(main_2,a,b)\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\} \\
(13) & \langle(l),(main_2,a,b)\rangle \hookrightarrow \langle(l),(lock_0)\,(main_3,a,b)\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\} \\
(14) & \langle(l),(main_3,a,b)\rangle \hookrightarrow \langle(l),(g_0,a)\,(main_{3'},a,b)\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\} \\
(15) & \langle(l,r),(main_{3'},a,b)\rangle \hookrightarrow \langle(l),(main_4,a,r)\rangle & a,b,l,r \in \{\mathtt{T},\mathtt{F}\} \\
(16) & \langle(l),(main_4,a,b)\rangle \hookrightarrow \langle(l),(unlock_0)\,(main_5,a,b)\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\} \\
(17) & \langle(l),(main_5,a,b)\rangle \hookrightarrow \langle(l),\varepsilon\rangle & a,b,l \in \{\mathtt{T},\mathtt{F}\}
\end{array}
$$

Figure 2.6: Pushdown rules for the program in Figure 2.4.

alphabet of the form $\Gamma = \Gamma_0 \times L$. Let

$$r := \langle p,\gamma\rangle \underset{[R]}{\hookrightarrow} \langle p',\gamma_1\ldots\gamma_n\rangle,$$

where $p,p' \in P_0,\ \gamma,\gamma_1,\ldots,\gamma_n \in \Gamma_0$, and $R \subseteq (G\times L)\times(G\times L^n)$ is a relation. We call r a *symbolic rule* which denotes the following set of pushdown rules:

$$\{\, \langle(p,g),(\gamma,l)\rangle \hookrightarrow \langle(p',g'),(\gamma_1,l_1)\ldots(\gamma_n,l_n)\rangle \mid (g,l,g',l_1,\ldots,l_n) \in R \,\}$$

Assuming that R usually has some compact representation, Δ can be succinctly represented as a list of symbolic rules. Somewhat informally, we call a pushdown system with such a data structure a *symbolic pushdown system.*

In this structure, Δ comes as a partitioned transitioned relation, and configurations are given in a mix of explicit and symbolic components; element of P_0 and Γ_0 are stated explicitly whereas elements of G and L are given symbolically. In the following, most of the theory about pushdown systems is developed independently of their actual representation, and the resulting algorithms can be instantiated appropriately. However, the usage of a partitioned representation does have certain practical advantages, as we shall see later.

What remains is to find good data structures for the symbolic representations. These need to satisfy certain demands: They must be compact in practical cases, and they must allow efficient set operations needed by the model-checking algorithms. The next section introduces Binary Decision Diagrams (BDDs) for this purpose. BDDs are particularly suited to represent sets of vectors of boolean values but can also be used to encode finite sets in general. In the experiments, only programs with boolean and integer variables are considered and BDDs are used to represent them. Integer variables with values from a finite range are simulated using multiple boolean variables. Other representations are conceivable and could be realised with only minor modifications of the algorithms in Chapter 3.

2.4 Binary Decision Diagrams

In the previous section we introduced the concept of a symbolic pushdown system in which the data-related part of the transition relation is stored as a set R. It is hoped that these sets (which are relations containing the admissible variable valuations before and after the application of pushdown rules) have, in practice, compact representations since the cost of enumerating their elements individually would make model-checking them infeasible. The representations used in this thesis are Boolean Decision Diagrams (BDDs). This section gives only a rough overview of BDDs; for more detailed treatments of the topic the reader is referred to, e.g., [2] or [11].

The basic idea behind BDDs is to equate sets with *boolean formulas*. Let us assume that the relations we want to represent contain only boolean variables. This is not a real restriction since we will only handle variables with a finite range anyway, and these can be simulated with multiple boolean variables. Let $V = \{v_1, \ldots, v_n\}$ be the variables occurring in one such relation.

The set of boolean formulas over V is defined as usual; each variable v_i (for $v_i \in V$) and the constants 'true' and 'false' are boolean formulas; if p and q are boolean formulas, then so are $\neg p$ and $p \vee q$. Given a valuation $\mathcal{V}\colon V \to \{0,1\}$ and a boolean formula p, we can evaluate whether $\mathcal{V}$ satisfies p (written $\mathcal{V} \models p$). For this purpose, negation ($\neg$) and disjunction ($\vee$) have the meaning they usually have in propositional logic and can be combined to build all other logical connectives.

Moreover, we can identify each formula p with the set $\{ \mathcal{V} \mid \mathcal{V} \models p \}$, i.e. the set of valuations that satisfy p; in fact we consider two formulas equal if they are identified with the same set. Conversely, we can denote a set by some corresponding formula. With this in mind, we can also express certain standard set operations by boolean operations; e.g., the union of the two sets denoted by the formulas p and q is the set associated with $p \vee q$.

Binary Decision Diagrams are designed to compactly represent boolean formulas. They were made popular by Bryant [11] and introduced into model-checking by McMillan [32]. Like other representations for the same purpose, BDDs have certain advantages and drawbacks, for instance that they can encode certain classes of sets more efficiently than others. (It is worth remembering at this point that by a simple counting argument no universal compression is possible, i.e. no encoding exists which compresses every single element of its domain by at least one bit.) However, BDDs are generally considered the most efficient method known to date, and ready-to-use, efficient libraries exist to handle them [38, 30]. In the experiments, we use the CUDD library by Fabio Somenzi [38].

Definition 2.8 *A BDD with domain V is a rooted directed acyclic graph whose nodes are labelled by elements of V or by the constants* 0 *and* 1*. Nodes labelled by constants are leaves with no outgoing edges; all other nodes haves two outgoing edges labelled by* 0 *and* 1*, respectively.*

The truth value of a BDD $\mathcal{B}$ for a valuation $\mathcal{V}\colon V \to \{0,1\}$ is determined as follows: Starting at the root node, we descend through the tree; if we are at a node labelled by $v \in V$, we follow the edge labelled by $\mathcal{V}(v)$; otherwise, if we are at a leaf, we terminate, and the truth value is given by the constant labelling of the leaf.

A BDD $\mathcal{B}$ represents a boolean formula p (or, equivalently, a set of valuations) in the following sense: $\mathcal{V} \models p$ if and only if the truth value of $\mathcal{V}$ in $\mathcal{B}$ is 1. Figure 2.7 shows a BDD with domain $\{a, b, a', b'\}$. The formula represented by the BDD is $(a' \leftrightarrow a \wedge b) \wedge (b \leftrightarrow b')$

We say that a BDD $\mathcal{B}$ is *ordered* if there exists an ordering $<$ on V such that if $\mathcal{B}$ has an edge from a node labelled v to a node labelled v', then $v < v'$. (In other words, the variables encountered when descending the diagram occur in the same order on every path.) The BDD in Figure 2.7 respects the ordering $a < b < a' < b'$.

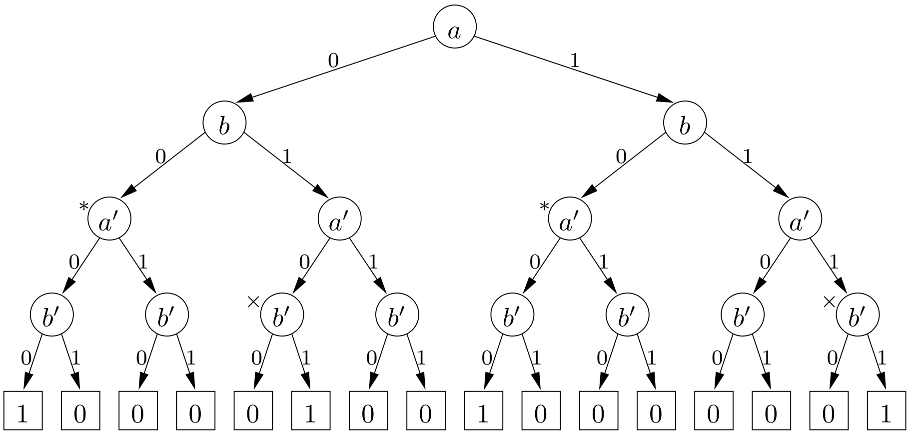

Figure 2.7: A BDD for the formula $(a' \leftrightarrow a \wedge b) \wedge (b \leftrightarrow b')$

More compact structures can be achieved with *reduced* BDDs. A BDD is said to be reduced under two conditions: It does not contain any two isomorphic subtrees, and it does not contain any non-leaf node whose two children are the same. Figure 2.8 shows a reduced BDD equivalent to the one from Figure 2.7. All terminal nodes have been combined to only two leaves labelled by 0 and 1. Moreover, Figure 2.7 contains two more isomorphic pairs of subtrees (marked there by $*$ and $\times$, respectively). In the reduced BDD, these are combined and occur only once.

An important feature of reduced ordered BDDs is their canonicity – for each boolean formula (and each ordering $<$) over V there exists one unique reduced ordered BDD. This makes it possible to check equivalence of two BDDs (or formulas, or sets) in constant time. Since the library used in the experiments uses reduced ordered BDDs, all references to BDDs in the following implicitly assume that the BDDs in question are, in fact, reduced and ordered BDDs. The size of a BDD representing a particular function (and hence the efficiency with which it can be processed) may very much depend on the ordering it adhers to. Ideas on how to derive a good ordering are discussed in Section 5.8.

2.5 Model checking and temporal logics

In order to provide a meaningful model-checking framework, we need a precise logic in which analysis problems can be stated. The most basic analysis

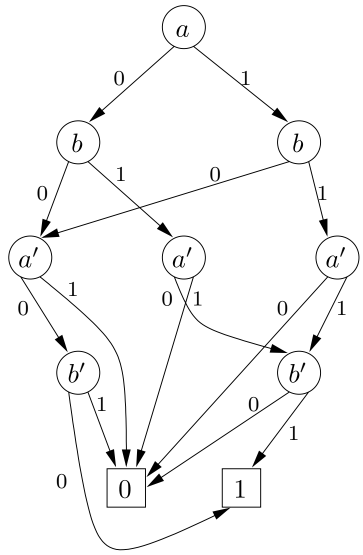

Figure 2.8: The BDD from Figure 2.7 in reduced form.

problem is the *reachability problem* which we can formulate in two variants:

(I) Given a transition system $\mathcal{T}$ and a state s, is s reachable in $\mathcal{T}$, i.e. is there a path from the root to s?

(II) Given a transition system $\mathcal{T}$, compute the set of all reachable states.

However, reachability cannot express many interesting properties such as "all runs of the process will eventually terminate" or "a certain state or action will occur infinitely often". This gap is filled by temporal logics in which properties like these can be expressed with formulas. A temporal logic framework defines the semantics of each formula as the set of states, paths, or runs which satisfy it. A model-checking problem then consists of checking whether some given states (or paths, or runs) satisfy a given property encoded in a formula of some temporal logic. More specifically, we can ask the following three questions:

(III) Given a transition system $\mathcal{T} = (S, A, \rightarrow, r)$ and a formula φ of some temporal logic, does the root r satisfy the formula φ (written $r \models \varphi$)?

(IV) Given a transition system $\mathcal{T}$ and a formula φ, compute the set of all states of $\mathcal{T}$ which satisfy (or violate) φ. This is also called the *global* model-checking problem.

(V) Given a transition system $\mathcal{T}$ and a formula φ, compute the set of all *reachable* states which satisfy (or violate) φ, i.e. the intersection of the solutions to problems (II) and (IV).

In this thesis, we will examine problems (I) to (V) for the cases where $\mathcal{T}$ is generated by a pushdown system (for general transition systems, all these questions are of course undecidable), for the temporal logic LTL and, as an extension, for the logic CTL*. We will also examine how computationally difficult these problems are.

Complexity measures The complexity of model-checking algorithms can be measured in different ways. We have two parameters to consider: the system and the temporal property. Therefore, we can state the complexity in three ways: As a function of the size of the system (i.e. for a fixed property), as a function of the size of the property (i.e. for a fixed system), or as a function of both. In practice the size of the system can be quite large whereas formulas are typically small. Therefore, the complexity in terms of the system can be regarded as the more important question.

The transition system associated with a pushdown system is actually infinitely large, but it is natural use the pushdown system itself as a basis for the complexity measure. In fact, we shall analyze the complexities in terms of parameters such as the number of control locations or the size of the stack alphabet.

As far as temporal logics are concerned, this section gives only some details relevant to the material in the following chapters; for a more thorough treatment of temporal logics see for instance [15]. The complexity of model-checking problems for various logics and different classes of transition systems including pushdown systems is treated in [31]. There are two main classes of temporal logics: Branching-time logics and linear-time logics.

2.5.1 Branching-time logics

Branching-time logics are interpreted over the *computation tree* of the transition system under consideration. The computation tree of a transition system $\mathcal{T} = (S, A, \rightarrow, r)$ is the (possibly infinite) tree whose nodes are labelled by elements of S, whose root is labelled with r, and whose edges are labelled with elements of A. If $s \xrightarrow{a} s'$ then a node labelled s has a child labelled s'

connected by an edge labelled a. The validity of formulas in branching-time logic is defined on the nodes of this computation tree; the truth value at each particular state depends on the state itself and the subtree below it.

Since branching-time logic will not play an important role in this thesis (for the reasons stated below), we will not present branching-time logics in any more detail; suffice to say that probably the best-known branching-time logic is CTL (Computation Tree Logic), introduced by Clarke and Emerson [13]. CTL's advantage over more expressive logics such as variants of the μ-calculus is that it is more intuitive to understand while still powerful enough to express many interesting properties. Moreover, CTL can be checked in linear time on finite state systems and is therefore widely used in this area. For pushdown systems, the situation is different; even for EF, a fragment of CTL, the model-checking problem is computationally intractable as shown by a result from [9].

Theorem 2.1 *There is formula φ of EF such that the problem of deciding if a pushdown system satisfies φ is PSPACE-complete.*

In other words, model-checking EF takes exponential time in the size of the pushdown system. Hence, efficient model-checking procedures for pushdown systems and CTL seem to be unattainable, even more so if we want to apply them to pushdown systems derived from programs for which the translation already involves one exponential blowup. Chapter 3 therefore concentrates on algorithms for linear-time logic. In Section 6.6 we show how these algorithms could (at least in theory) be used to extend the logic to CTL*, a superset of CTL.

2.5.2 Linear-time logics

In linear-time logic the semantics of each formula is given as a set of ω-words. The most widely used variant of this kind is Linear-Time Temporal Logic (LTL), introduced by Pnueli [34]:

Let At be a (countable) set of atomic propositions. LTL formulas are built according to the following abstract syntax equation (where A ranges over At):

$$\varphi \;::=\; \mathtt{tt} \mid A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

$\mathcal{X}$ and $\mathcal{U}$ are called the *next* and *until* operators, respectively. Let $At(\varphi)$ be the set of atomic propositions which appear in φ (note that $At(\varphi)$ is finite).

Each formula φ defines a language $L(\varphi)$ of ω-words over the alphabet $2^{At(\varphi)}$ that satisfy φ; we denote that ω satisfies φ by $\omega \models \varphi$. The satisfaction relation is defined inductively on the structure of φ as follows:

$$
\begin{array}{lcl}
\omega \models \mathtt{tt} & & \\
\omega \models A & \iff & A \in \omega(0) \\
\omega \models \neg\varphi & \iff & \omega \not\models \varphi \\
\omega \models \varphi_1 \wedge \varphi_2 & \iff & \omega \models \varphi_1 \text{ and } \omega \models \varphi_2 \\
\omega \models \mathcal{X}\varphi & \iff & \omega^1 \models \varphi \\
\omega \models \varphi_1 \mathcal{U} \varphi_2 & \iff & \exists i\colon (\omega^i \models \varphi_2) \wedge (\forall j < i\colon \omega^j \models \varphi_1)
\end{array}
$$

We also define $\Diamond\varphi \equiv \mathtt{tt}\,\mathcal{U}\,\varphi$ and $\Box\varphi \equiv \neg(\Diamond\neg\varphi)$. The $\Diamond\varphi$ operator ("eventually φ") expresses that ϕ will hold at some point in the future, whereas the $\Box\varphi$ operator ("always φ") states that φ holds in every step from now on.

LTL can be interpreted on a transition system $\mathcal{T} = (S, A, \rightarrow, r)$ by means of a valuation function. A valuation function assigns to each atomic proposition the set of states or actions that satisfy it. Thus, each run of the system can be translated into an ω-word over the alphabet $2^{At(\varphi)}$ containing the atomic propositions that hold at each point of time during the run.

Since a run σ is labelled by an infinite sequence of actions ω, the logics can be interpreted in two ways: state-based or action-based. In the following we shall use and present algorithms for the state-based semantics; however, this choice is not essential, and action-based variants could be achieved with only minor modifications. A valuation is therefore a function $\nu\colon At(\varphi) \rightarrow S$ and the translation σ_ν is given by $\sigma_\nu(i) = \{\, A \mid A \in At(\varphi),\ \sigma(i) \in \nu(A) \,\}$ for all $i \geq 0$. The run σ satisfies φ under the valuation ν if $\sigma_\nu \models \varphi$. We say that a state $s \in S$ satisfies φ under the valuation ν, written $s \models^\nu \varphi$, if and only if all runs starting at s satisfy φ. We say that $\mathcal{T}$ satisfies φ (written $\mathcal{T} \models^\nu \varphi$) if and only if $r \models^\nu \varphi$.

Examples The termination property above ("all runs of the process will eventually terminate") can be expressed in LTL as

$$\Diamond \mathit{term}$$

if the atomic proposition *term* is true of all states in which the process has terminated. The property that proposition p will occur infinitely often in all executions is expressed by

$$\Box\Diamond p$$

In the plotter example (see Figures 2.1 through 2.3), a desired correctness property could be that an upward movement should never be immediately followed by a downward movement and vice versa:

$$\Box(up \rightarrow (\neg down\,\mathcal{U}\,(up \vee right))) \;\wedge\; \Box(down \rightarrow (\neg up\,\mathcal{U}\,(down \vee right)))$$

In the pushdown system corresponding to the program, the atomic proposition *up* should be true of configurations which correspond to the entry point of procedure *up*, i.e. those of the forms $\langle\,\cdot\,, up_0\, w\rangle$. Similarly, *down* should be true of configurations with $down_0$ as the topmost stack symbol, and *right* of those with $right_0$.

Some well-known properties of data-flow analysis (e.g., liveness, reachability, very business, availability) can be expressed in LTL. For instance, a program variable Y is said to be *dead* at a program point n if in every possible continuation from n we have that Y is either not used or is redefined before it is used. Suppose that we are given a pushdown system derived from the control-flow graph of a program (see Section 2.3). If we want to know at which program points Y is dead, we can model-check the formula

$$(\neg used_Y\,\mathcal{U}\,def_Y) \;\vee\; (\Box\neg used_Y)$$

where $used_Y$, and def_Y are atomic propositions which are valid in exactly those configurations where the topmost stack symbol corresponds to an instruction which uses the variable Y, or to an instruction which defines Y, respectively. The solution of problem (IV) in this case yields the set of all program configurations from which there *is* a run in which the value of Y is used before it is redefined (i.e. Y is *not* dead in these configurations).

Reachability can be expressed only indirectly. If c is the configuration whose reachability we want to check, we introduce a single atomic proposition p_c which is true only for c. Then the following formula expresses that c is *unreachable*:

$$\Box\neg p_c$$

Complexity measures (continued) The following well-known theorem (see, e.g., [41]) will play a central role in the model-checking algorithms:

Theorem 2.2 *Given an LTL formula φ, one can effectively construct a Büchi automaton $\mathcal{B}_\varphi$ of the form $\mathcal{B}_\varphi = (Q, 2^{At(\varphi)}, \rightarrow, q_0, F)$ of size $\mathcal{O}(2^{|\varphi|})$ which recognizes the language $L(\varphi)$.*

Our solution to LTL model-checking (see Section 3.2) involves translating an LTL formula φ into a Büchi automaton $\mathcal{B}$, and the complexity of the solution depends on the size of the automaton. In general, $\mathcal{B}$ may be exponentially larger than φ; however, this is often not the case. Therefore it seems to make sense to measure complexity in terms of $\mathcal{B}$, and our results will be stated in this way. The worst-case complexity in terms of φ is therefore exponential, but this is unavoidable according to a result of Bouajjani et al [9]:

Theorem 2.3 *The model-checking problem for LTL and pushdown systems is DEXPTIME-complete.*

Chapter 3

Model-Checking Algorithms

In this chapter we define the problems we study as well as algorithms for their solutions. Before starting, we introduce a restriction that will make the presentation slightly easier: We restrict the problems to pushdown systems which satisfy $|w| \leq 2$ for every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$. This restriction is natural in light of the intended application (see Section 2.3) where the stack is used to store activation records and can increase by at most one in a single transition. Moreover, the restriction is not truly restrictive since by the following Theorem 3.1 any pushdown system can be converted into such a normal form at the cost of introducing additional configurations:

Definition 3.1 *Let $\mathcal{P}$ and $\mathcal{P}'$ be pushdown systems. $\mathcal{P}'$* simulates *$\mathcal{P}$ if $Conf(\mathcal{P}) \subseteq Conf(\mathcal{P}')$ and the following condition holds: for any two configurations $c, c' \in Conf(\mathcal{P})$ we have $c \Rightarrow_{\mathcal{P}} c'$ if and only if $c \underset{n}{\Longrightarrow}_{\mathcal{P}'} c'$ for some $n \geq 1$ and all the $n-1$ configurations between c and c' are in $Conf(\mathcal{P}') \setminus Conf(\mathcal{P})$.*

Theorem 3.1 *Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$, we can construct a pushdown system $\mathcal{P}' = (P, \Gamma', \Delta', c_0)$ such that*

(a) $|w| \leq 2$ holds for each rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta'$;

(b) $\mathcal{P}'$ simulates $\mathcal{P}$;

(c) $|\mathcal{P}'| = \mathcal{O}(|\mathcal{P}|)$.

Proof:

(a) Let Γ' and Δ' be the least sets such that

1. $\Gamma \subseteq \Gamma'$;
2. if Δ has a rule r of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ such that $|w| \leq 2$, then $r \in \Delta'$;
3. if Δ has a rule r of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \dots \gamma_n \rangle$ such that $n > 2$, then $\{\delta_{r,1}, \dots, \delta_{r,n-2}\} \subseteq (\Gamma' \setminus \Gamma)$ and Δ' contains the rules

   – $\langle p, \gamma \rangle \hookrightarrow \langle p', \delta_{r,n-2}\gamma_n \rangle$
   – $\langle p', \delta_{r,i} \rangle \hookrightarrow \langle p', \delta_{r,i-1}\gamma_{i+1} \rangle$ for $2 \leq i \leq n-2$
   – $\langle p', \delta_{r,1} \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle$

(b) Suppose that $c \Rightarrow_{\mathcal{P}} c'$. Then $c \underset{n}{\Longrightarrow}_{\mathcal{P}'} c'$ obviously holds because the effect of the pushdown rule that was used to get from c to c' in $\mathcal{P}$ can be achieved by one or more rules in $\mathcal{P}'$. For the other direction, note that every element of $\Gamma' \setminus \Gamma$ occurs on the left-hand side of exactly one rule, thus every configuration whose topmost stack symbol is from $\Gamma' \setminus \Gamma$ has only one immediate successor. Hence, if a path in $\mathcal{P}'$ starts at $c \in \mathit{Conf}(\mathcal{P})$ and goes to some configuration in $\mathit{Conf}(\mathcal{P}') \setminus \mathit{Conf}(\mathcal{P})$, the rest of the path is uniquely determined until the next $c' \in \mathit{Conf}(\mathcal{P})$ is reached. An inspection of the rules in $\Delta' \setminus \Delta$ shows that c' is an immediate successor of c in $\mathcal{P}$.

(c) For every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \dots \gamma_n \rangle \in \Delta$, Γ' contains no more than $n-2$ additional symbols, and Δ' contains no more than $n-1$ rules of constant size. Thus, $|\Gamma'| = \mathcal{O}(|\Gamma| + |\Delta|)$, $|\Delta'| = \mathcal{O}(|\Delta|)$, and therefore $|\mathcal{P}'| = \mathcal{O}(|\mathcal{P}|)$.

□

For these reasons, all problems and algorithms will be presented for pushdown systems which satisfy the aforementioned restriction. If, for some reason, we want to solve one of the analysis problems for an unrestricted pushdown system $\mathcal{P}$, we can apply the construction from Theorem 3.1 to get a restricted system $\mathcal{P}'$ and solve the problem for $\mathcal{P}'$. Since the additional configurations of $\mathcal{P}'$ are easily recognisable, simple procedures exist to remove them from the result of the algorithms. Since the asymptotic sizes of $\mathcal{P}$ and $\mathcal{P}'$ are equal, the complexity results we derive also hold for unrestricted pushdown systems.

3.1 Reachability

Reachability analysis comes in two flavours: forward and backward. Given a pushdown system $\mathcal{P}$ and a set $C \subseteq Conf(\mathcal{P})$, a backward reachability analysis consists of computing the predecessors of elements of C, i.e. the set $pre^*(C)$. The forward reachability analysis consists of computing the successors of elements of C, i.e. the set $post^*(C)$.

In general, C can be an infinite set. Even if C is finite, however, the sets $pre^*(C)$ or $post^*(C)$ may be infinite. Therefore, we cannot hope to represent sets of configurations explicitly – we need a finite symbolic representation for them. In the following, we present algorithms to solve both problems for the case where C is a regular language (in a sense made precise by the following definition).

Definition 3.2 *Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system. A $\mathcal{P}$-*automaton *uses Γ as alphabet, and P as its set of initial states (we consider automata with possibly many initial states). Formally, a $\mathcal{P}$-automaton is a quintuple $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the set of* transitions*, and $F \subseteq Q$ the set of* final states. *$\mathcal{P}$* accepts *or* recognizes *a configuration $\langle p, w \rangle$ if the transition system $(Q, \Gamma, \rightarrow, p)$ satisfies $p \xrightarrow{w}{}^* q$ for some $q \in F$. The set of configurations recognised by a $\mathcal{P}$-automaton $\mathcal{A}$ is denoted by $L(\mathcal{A})$. A set of configurations of $\mathcal{P}$ is* regular *if it is recognized by some $\mathcal{P}$-automaton.*

It is important to understand that a $\mathcal{P}$-automaton simply represents a set of configurations of $\mathcal{P}$, and not to confuse its behaviour with that of $\mathcal{P}$.

In order to handle the problems (I) to (V) presented in Section 2.5 it will be enough to solve the reachability analysis problems for regular sets of configurations. For problems (I) and (II) we can, for instance, compute $post^*(C)$ for the regular singleton set $C = \{c_0\}$. For problems (III) to (V), the pre^* and $post^*$ operations will be important building blocks.

Notation Fix a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ for the rest of the section. Since all the automata we encounter in this section are $\mathcal{P}$-automata, we drop the $\mathcal{P}$ from now on. We use the symbols p, p', p'' etc., eventually with indices, to denote initial states of an automaton (i.e., the elements of P). Non-initial states are denoted by s, s', s'' etc., and arbitrary states, initial or not, by q, q', q''.

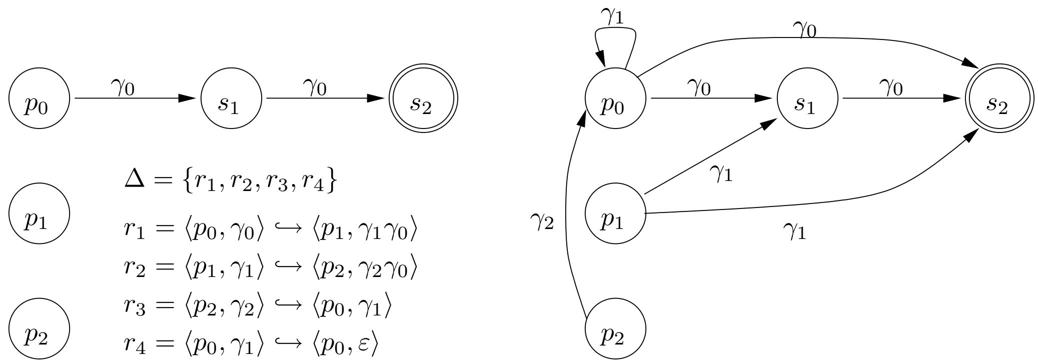

Figure 3.1: The automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$ (right)

The algorithms for pre^* and $post^*$ are introduced in two steps. First, abstract procedures are shown which are relatively easy to understand. Then we present efficient algorithms and prove that they are correct implementations of the abstract procedures.

3.1.1 Computing pre^*

Our input is an automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ accepting a regular set C. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We compute $pre^*(C)$ as the language accepted by an automaton $\mathcal{A}_{pre^*} = (Q, \Gamma, \rightarrow, P, F)$ obtained from $\mathcal{A}$ by means of a saturation procedure. The procedure adds new transitions to $\mathcal{A}$, but no new states. New transitions are added according to the following *saturation rule*:

If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w}{}^* q$ in the current automaton, add a transition (p, γ, q).

The saturation procedure eventually reaches a fixed point because the number of possible new transitions is finite.

The intuition behind this procedure is that if there is a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, then $\langle p, \gamma w' \rangle$ is a predecessor of $\langle p', ww' \rangle$. Thus, if the automaton already accepts $\langle p', ww' \rangle$ by way of $p \xrightarrow{w}{}^* q \xrightarrow{w'}{}^* q_f$ (for some final state q_f), then it also ought to accept $\langle p, \gamma w' \rangle$. Adding the transition (p, γ, q) achieves this by adding the sequence $p \xrightarrow{\gamma} q \xrightarrow{w'}{}^* q_f$.

Example: Let us illustrate the procedure by an example. Consider the pushdown system with control locations $P = \{p_0, p_1, p_2\}$ and Δ as shown in in the left half of Figure 3.1. Let $\mathcal{A}$ be the automaton that accepts the set $C = \{\langle p_0, \gamma_0\gamma_0\rangle\}$, also shown on the left. The result of the algorithm is shown in the right half of Figure 3.1. The result is derived through the following six steps:

1. First, we note that $p_0 \xrightarrow{\varepsilon}{}^{*} p_0$ holds. Since $\langle p_0, \varepsilon\rangle$ occurs on the right-hand side of rule r_4, we have a match for the saturation rule, and we add the transition (p_0, γ_1, p_0).

2. Now that we have $p_0 \xrightarrow{\gamma_1} p_0$, there is a match for the saturation rule with r_3, and so we add (p_2, γ_2, p_0).

3. This in turn creates the path $p_2 \xrightarrow{\gamma_2} p_0 \xrightarrow{\gamma_0} s_1$ which, because of r_2, leads to the addition of (p_1, γ_1, s_1).

4. The rule r_1 in conjunction with $p_1 \xrightarrow{\gamma_1\gamma_0}{}^{*} s_2$ give us (p_0, γ_0, s_2).

5. The previous addition creates a path $p_2 \xrightarrow{\gamma_2\gamma_0}{}^{*} s_2$ which allows another application of rule r_2 leading to (p_1, γ_1, s_2).

6. No further additions are possible, so the procedure terminates.

Proof: Let $\mathcal{A}$ be a $\mathcal{P}$-automaton, and let $\mathcal{A}_{pre^*}$ be the automaton obtained from $\mathcal{A}$ by means of the saturation rule defined above. We show that $\mathcal{A}_{pre^*}$ recognises the set $pre^*(L(\mathcal{A}))$. In the sequel we use $\rightarrow_i$ to denote the transition relation in the automaton obtained after adding i transitions to $\mathcal{A}$; in particular, $\rightarrow_0$ is the transition relation of $\mathcal{A}$. Moreover, $\rightarrow$ (the transition relation of $\mathcal{A}_{pre^*}$) subsumes all $\rightarrow_i$. Observe that all new transitions added by the procedure start at an initial state, i.e. an element of P. The result is proved in Theorem 3.2 at the end of this subsection. We need two preliminary lemmata.

Lemma 3.1 *For every configuration $\langle p, v\rangle \in L(\mathcal{A})$, if $\langle p', w\rangle \Rightarrow^* \langle p, v\rangle$ then $p' \xrightarrow{w}{}^{*} q$ for some final state q of $\mathcal{A}_{pre^*}$.*

Proof: Assume $\langle p', w\rangle \underset{k}{\Longrightarrow} \langle p, v\rangle$. We proceed by induction on k.

Basis. $k = 0$. Then $p' = p$ and $w = v$. Since $\langle p, v\rangle \in L(\mathcal{A})$, we have $p \xrightarrow{v}{}_0^{*} q$ for some final state q, and so $p \xrightarrow{v}{}^{*} q$, which implies $p' \xrightarrow{w}{}^{*} q$.

Step. $k > 0$. Then there is a configuration $\langle p'', u \rangle$ such that

$$\langle p', w \rangle \underset{1}{\Longrightarrow} \langle p'', u \rangle \underset{k-1}{\Longrightarrow} \langle p, v \rangle \ .$$

We apply the induction hypothesis to $\langle p'', u \rangle \underset{k-1}{\Longrightarrow} \langle p, v \rangle$, and obtain

$$p'' \xrightarrow{u}{}^{*} q \text{ for some } q \in F \ .$$

Since $\langle p', w \rangle \underset{1}{\Longrightarrow} \langle p'', u \rangle$, there are γ, w_1, u_1 such that

$$w = \gamma w_1,\ u = u_1 w_1, \text{ and } \langle p', \gamma \rangle \hookrightarrow \langle p'', u_1 \rangle \in \Delta.$$

Let q_1 be a state of $\mathcal{A}_{pre^*}$ such that

$$p'' \xrightarrow{u_1}{}^{*} q_1 \xrightarrow{w_1}{}^{*} q.$$

By the saturation rule, we have

$$p' \xrightarrow{\gamma} q_1 \xrightarrow{w_1}{}^{*} q \ ,$$

since $w = \gamma w_1$, this implies

$$p' \xrightarrow{\gamma w_1}{}^{*} q \ .$$

□

Lemma 3.2 *If $p \xrightarrow{w}{}^{*} q$, then the following properties hold:*

(a) $\langle p, w \rangle \Rightarrow^* \langle p', w' \rangle$ *for a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'}{}^{*}_{0} q$;*

(b) *moreover, if q is an initial state, then $w' = \varepsilon$.*

Proof:

Let i be an index such that $p \xrightarrow{w}{}^{*}_{i} q$ holds. We shall prove (a) by induction on i. Statement (b) then follows immediately from the fact that initial states have no incoming transitions in $\mathcal{A}$.

Basis. $i = 0$. Since $\langle p, w \rangle \Rightarrow^* \langle p, w \rangle$ always holds, take $p' = p$ and $w' = w$.

Step. $i \geq 1$. Let $t = (p_1, \gamma, q')$ be the i-th transition added to $\mathcal{A}$. (Notice that we can safely write (p_1, γ, q') instead of (q_1, γ, q') because all new transitions start at an initial state.) Let j be the number of times that t is used in $p \xrightarrow{w}_i^* q$.

The proof is by induction on j. If $j = 0$, then we have $p \xrightarrow{w}_{i-1}^* q$, and (a) follows from the induction hypothesis (induction on i). So assume that $j > 0$. Then there exist u and v such that $w = u\gamma v$ and

$$p \xrightarrow{u}_{i-1}^* p_1 \xrightarrow{\gamma}_i q' \xrightarrow{v}_i^* q \tag{0}$$

The application of the induction hypothesis (induction on i) to $p \xrightarrow{u}_{i-1}^* p_1$ yields (notice that p_1 is an initial state, and so both (a) and (b) can be applied):

$$\langle p, u \rangle \Rightarrow^* \langle p_1, \varepsilon \rangle \tag{1}$$

Since the transition (p_1, γ, q') has been added by applying the saturation rule, there exist p_2 and w_2 such that

$$\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, w_2 \rangle \tag{2.1}$$
$$p_2 \xrightarrow{w_2}_{i-1}^* q' \tag{2.2}$$

From (0) and (2.2) we get

$$p_2 \xrightarrow{w_2}_{i-1}^* q' \xrightarrow{v}_i^* q \tag{3}$$

Since the transition t is used in (3) less often than in (0), we can apply the induction hypothesis (induction on j) to (3), and obtain

$$\langle p_2, w_2 v \rangle \Rightarrow^* \langle p', w' \rangle \tag{4.1}$$
$$p' \xrightarrow{w'}_0^* q \tag{4.2}$$

Putting (1), (2.1), and (4.1) together, we get

$$\langle p, w \rangle = \langle p, u\gamma v \rangle \Rightarrow^* \langle p_1, \gamma v \rangle \Rightarrow \langle p_2, w_2 v \rangle \Rightarrow^* \langle p', w' \rangle \tag{5}$$

We obtain the claim from (5) and (4.2).

□

Theorem 3.2 *Let $\mathcal{A}_{pre^*}$ be the automaton obtained from $\mathcal{A}$ by exhaustive application of the saturation rule defined in the beginning of Section 3.1.1. $\mathcal{A}_{pre^*}$ recognises the set $pre^*(L(\mathcal{A}))$.*

Proof: Let $\langle p, w\rangle$ be a configuration of $pre^*(L(\mathcal{A}))$. Then $\langle p, w\rangle \Rightarrow^* \langle p', w'\rangle$ for a configuration $\langle p', w'\rangle \in L(\mathcal{A})$. By Lemma 3.1, $p \xrightarrow{w}{}^* q$ for some final state q of $\mathcal{A}_{pre^*}$. So $\langle p, w\rangle$ is recognised by $\mathcal{A}_{pre^*}$.

Conversely, let $\langle p, w\rangle$ be a configuration accepted by $\mathcal{A}_{pre^*}$. Then $p \xrightarrow{w}{}^* q$ in $\mathcal{A}_{pre^*}$ for some final state q. By Lemma 3.2(a), $\langle p, w\rangle \Rightarrow^* \langle p', w'\rangle$ for a configuration $\langle p', w'\rangle$ such that $p' \xrightarrow{w'}{}^*_0 q$. Since q is a final state, we have $\langle p', w'\rangle \in L(\mathcal{A})$, and so $\langle p, w\rangle \in pre^*(L(\mathcal{A}))$. $\square$

3.1.2 Computing *post**

Again, our input is an automaton $\mathcal{A}$ accepting C. Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state. We compute $post^*(C)$ as the language accepted by an automaton $\mathcal{A}_{post^*}$ with ε-moves. $\mathcal{A}_{post^*}$ is obtained from $\mathcal{A}$ in two stages:

- For each pair (p', γ') such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$, add a new state $q_{p',\gamma'}$.
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

(i)	If $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition (p', ε, q).
(ii)	If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition (p', γ', q).
(iii)	If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, first add $(p', \gamma', q_{p',\gamma'})$ and then $(q_{p',\gamma'}, \gamma'', q)$.

Example: Consider again the pushdown system $\mathcal{P}$ and the automaton $\mathcal{A}$ from Figure 3.1. Then the automaton shown in Figure 3.2 is the result of the new algorithm and accepts $post^*(\{\langle p_0, \gamma_0\gamma_0\rangle\})$. The result is reached via the following steps:

1. First, we add the new states: $r_1 = \langle p_0, \gamma_0\rangle \hookrightarrow \langle p_1, \gamma_1\gamma_0\rangle$ leads to q_{p_1,γ_1} and $r_2 = \langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_0\rangle$ to q_{p_2,γ_2}.

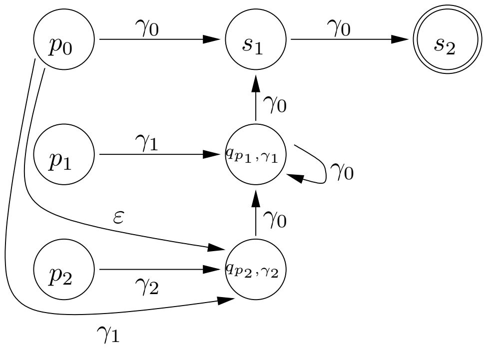

Figure 3.2: $\mathcal{A}_{post^*}$

2. Since $p_0 \xrightarrow{\gamma_0} s_1$ matches the left-hand side of rule r_1, the first transitions we add are $(p_1, \gamma_1, q_{p_1,\gamma_1})$ and $(q_{p_1,\gamma_1}, \gamma_0, s_1)$.
3. The first of these new transitions together with rule r_2 gives rise to two more transitions: $(p_2, \gamma_2, q_{p_2,\gamma_2})$ and $(q_{p_2,\gamma_2}, \gamma_0, q_{p_1,\gamma_1})$.
4. Because of $(p_2, \gamma_2, q_{p_2,\gamma_2})$ and r_3 the next step is to add $(p_0, \gamma_1, q_{p_2,\gamma_2})$.
5. This in turn, together with r_4 leads to the ε-edge from p_0 to q_{p_2,γ_2}.
6. We now have $p_0 \xrightarrow{\gamma_0}{}^* q_{p_1,\gamma_1}$ and can apply rule r_1 once more. Because $(p_1, \gamma_1, q_{p_1,\gamma_1})$ has been added before, we just add $(q_{p_1,\gamma_1}, \gamma_0, q_{p_1,\gamma_1})$.
7. No unprocessed matches remain, so the procedure terminates.

Proof: Let $\mathcal{A}$ be a $\mathcal{P}$-automaton, and let $\mathcal{A}_{post^*}$ be the automaton obtained from $\mathcal{A}$ by means of the saturation rules. We show that $\mathcal{A}_{post^*}$ recognizes the set $post^*(L(\mathcal{A}))$. In the sequel we use a notation similar to the proof for pre^*; the transition relation of $\mathcal{A}_{post^*}$ is $\rightarrow$, the relation of the automaton after i transitions have been added is denoted by $\rightarrow_i$. The result is proved in Theorem 3.3 at the end of this subsection. We need two preliminary lemmata.

Lemma 3.3 *For every configuration $\langle p, v\rangle \in L(\mathcal{A})$, if $\langle p, v\rangle \Rightarrow^* \langle p', w\rangle$ then $p' \xrightarrow{w}{}^* q$ for some final state q of $\mathcal{A}_{post^*}$.*

Proof: Assume $\langle p, v\rangle \overset{}{\underset{k}{\Longrightarrow}} \langle p', w\rangle$. We proceed by induction on k.

Basis. $k = 0$. Then $p' = p$ and $w = v$. Since $\langle p, v\rangle \in L(\mathcal{A})$, we have $p \xrightarrow{v}_0^* q$ for some final state q, and so $p \xrightarrow{v}^* q$, which implies $p' \xrightarrow{w}^* q$.

Step. $k > 0$. Then there is a configuration $\langle p'', u\rangle$ with

$$\langle p, v\rangle \xRightarrow[k-1]{} \langle p'', u\rangle \xRightarrow[1]{} \langle p', w\rangle \ .$$

We apply the induction hypothesis to $\langle p, v\rangle \xRightarrow[k-1]{} \langle p'', u\rangle$, and obtain

$$p'' \xrightarrow{u}^* q \text{ for some } q \in F \ .$$

Since $\langle p'', u\rangle \xRightarrow[1]{} \langle p', w\rangle$, there are γ, u_1, w_1 such that

$$u = \gamma u_1,\ w = w_1 u_1, \text{ and } \langle p'', \gamma\rangle \hookrightarrow \langle p', w_1\rangle \in \Delta.$$

There are three possible cases distinguished by the length of w_1. We consider only the case $|w_1| = 2$, the others being simpler. Since $|w_1| = 2$, we have $w_1 = \gamma'\gamma''$. Let q_1 be a state of $\mathcal{A}_{post^*}$ such that

$$p'' \xrightarrow{\gamma}^* q_1 \xrightarrow{u_1}^* q.$$

By the saturation rule, we have

$$p' \xrightarrow{\gamma'} q_{p',\gamma'} \xrightarrow{\gamma''} q_1 \xrightarrow{u_1}^* q \ ,$$

and since $w = w_1 u_1$, this implies

$$p' \xrightarrow{w_1 u_1}^* q \ .$$

$\square$

Lemma 3.4 *If $p \xrightarrow{w}^* q$, then the following holds:*

(a) if q is a state of $\mathcal{A}$, then $\langle p', w'\rangle \Rightarrow^* \langle p, w\rangle$ for a configuration $\langle p', w'\rangle$ such that $p' \xrightarrow{w'}_0^* q$;

(b) if $q = q_{p',\gamma'}$ is a new state, then $\langle p', \gamma'\rangle \Rightarrow^* \langle p, w\rangle$.

Proof: Let i be an index such that $p \xrightarrow{w}_i^* q$. We prove both parts of the lemma by induction on i.

Basis. $i = 0$. Only (a) applies. Take $p' = p$ and $w' = w$.

Step. $i \geq 1$. Let t be the i-th transition added to $\mathcal{A}$. Let j be the number of times that t is used in $p \xrightarrow{w}_i^* q$. $\mathcal{A}$ has no transitions leading to initial states, and the algorithm does not add any such transitions; therefore, if t starts in an initial state, t can be used at most once and only at the start of the path.

The proof is by induction on j. If $j = 0$, then we have $p \xrightarrow{w}_{i-1}^* q$, and we apply the induction hypothesis (induction on i). So assume that $j > 0$. We distinguish three possible cases:

1. If t was added by cases (i) or (ii) of the saturation rule, then $t = (p_1, v, q_1)$, where $v = \varepsilon$ or $v = \gamma_1$. Then $j = 1$ and there exists w_1 such that $w = vw_1$ and q_1 such that

$$p = p_1 \xrightarrow{v}_i^* q_1 \xrightarrow{w_1}_{i-1}^* q \tag{0}$$

Because of the saturation rule, there exist p_2 and γ_2 such that

$$\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, v \rangle \tag{1.1}$$

$$p_2 \xrightarrow{\gamma_2}_{i-1}^* q_1 \tag{1.2}$$

From (0) and (1.2) we get

$$p_2 \xrightarrow{\gamma_2}_{i-1}^* q_1 \xrightarrow{w_1}_{i-1}^* q \tag{2}$$

As t is not used in (2), we apply the induction hypothesis (on j). For (a), we obtain

$$\langle p', w' \rangle \Rightarrow^* \langle p_2, \gamma_2 w_1 \rangle \tag{3.1}$$

$$p' \xrightarrow{w'}_0^* q \tag{3.2}$$

Combining (3.1) and (1.1) we have

$$\langle p', w' \rangle \Rightarrow^* \langle p_2, \gamma_2 w_1 \rangle \Rightarrow \langle p_1, v w_1 \rangle = \langle p, w \rangle \tag{4}$$

Part (a) of the lemma follows from (3.2) and (4). For (b) we assume that $q = q_{p',\gamma'}$; the induction hypothesis applied to (2) gives us

$$\langle p', \gamma' \rangle \Rightarrow^* \langle p_2, \gamma_2 w_1 \rangle \tag{5}$$

Combining (5) and (1.1) yields the proof of part (b):

$$\langle p', \gamma' \rangle \Rightarrow^* \langle p_2, \gamma_2 w_1 \rangle \Rightarrow \langle p_1, v w_1 \rangle = \langle p, w \rangle \tag{6}$$

2. If t is the first transition added in case (iii) of the saturation rule, then it has the form $(p', \gamma', q_{p',\gamma'})$. Assume that the transition is indeed new (otherwise we just apply the induction hypothesis on i). Then, since $q_{p',\gamma'}$ has no transitions leading into it initially, it cannot have played part in an application rule before this step, and t is the first transition leading to it. Also, there are no transitions leading away from t so far. So the only path using t is $p' \xrightarrow{\gamma'}_i q_{p',\gamma'}$. For this path we only need to prove part (b), and $\langle p', \gamma' \rangle \Rightarrow^* \langle p', \gamma' \rangle$ holds trivially.

3. Let $t = (q_{p_1,\gamma_1}, \gamma'', q')$ be the second transition added by the third part of the saturation rule. Then there exist u, v such that $w = u\gamma''v$ and

$$p \xrightarrow{u}{}^*_{i-1} q_{p_1,\gamma_1} \xrightarrow{\gamma''}_i q' \xrightarrow{v}{}^*_i q \tag{7}$$

Because t was added via the saturation rule, we conclude that there exists some rule of the form

$$\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma'' \rangle \tag{8.1}$$

$$p_2 \xrightarrow{\gamma_2}{}^*_{i-1} q' \tag{8.2}$$

Application of the induction hypothesis (on i) yields

$$\langle p_1, \gamma_1 \rangle \Rightarrow^* \langle p, u \rangle \tag{9}$$

Combining (7) and (8.2) we get

$$p_2 \xrightarrow{\gamma_2}{}^*_{i-1} q' \xrightarrow{v}{}^*_i q$$

Since t occurs less often than j in this path, we can apply the induction hypothesis (on j). For (a) we obtain the existence of some $\langle p', w' \rangle$ such that

$$\langle p', w' \rangle \Rightarrow^* \langle p_2, \gamma_2 v \rangle \tag{10.1}$$

$$p' \xrightarrow{w'}{}^*_0 q \tag{10.2}$$

If we put together (10.1), (8.1) and (9), we get

$$\langle p', w' \rangle \Rightarrow^* \langle p_2, \gamma_2 v \rangle \Rightarrow \langle p_1, \gamma_1 \gamma'' v \rangle \Rightarrow^* \langle p, u\gamma'' v \rangle = \langle p, w \rangle \tag{11}$$

and (a) follows from (10.2) and (11). For (b) assume that $q = q_{p',\gamma'}$ and the induction hypothesis (on j) yields

$$\langle p', \gamma' \rangle \Rightarrow^* \langle p_2, \gamma_2 v \rangle \tag{11}$$

Part (b) then follows from (11), (8.1) and (9):

$$\langle p', \gamma' \rangle \Rightarrow^* \langle p_2, \gamma_2 v \rangle \Rightarrow \langle p_1, \gamma_1 \gamma'' v \rangle \Rightarrow^* \langle p, u\gamma'' v \rangle = \langle p, w \rangle \tag{12}$$

$\square$

Theorem 3.3 *Let $\mathcal{A}_{post^*}$ be the automaton obtained from $\mathcal{A}$ by exhaustive application of the saturation rule defined at the beginning of Section 3.1.2. $\mathcal{A}_{post^*}$ recognises the set $post^*(L(\mathcal{A}))$.*

Proof:

Let $\langle p, w \rangle$ be a configuration of $post^*(L(\mathcal{A}))$. Then $\langle p', w' \rangle \Rightarrow^* \langle p, w \rangle$ for a configuration $\langle p', w' \rangle \in L(\mathcal{A})$. By Lemma 3.3, $p \xrightarrow{w}{}^* q$ for some final state q of $\mathcal{A}_{post^*}$. So $\langle p, w \rangle$ is recognised by $\mathcal{A}_{post^*}$.

Conversely, let $\langle p, w \rangle$ be a configuration recognised by $\mathcal{A}_{post^*}$. Then $p \xrightarrow{w}{}^* q$ in $\mathcal{A}_{post^*}$ for some final state q. Since the construction does not add final states, q is a state of $\mathcal{A}$. By Lemma 3.4, $\langle p', w' \rangle \Rightarrow^* \langle p, w \rangle$ for a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'}{}^*_0 q$. Since q is a final state, $\langle p', w' \rangle \in L(\mathcal{A})$, and so $\langle p, w \rangle \in post^*(L(\mathcal{A}))$. $\square$

3.1.3 An efficient implementation for pre^*

We now present a concrete implementation of the abstract algorithm from Section 3.1.1. Again, given an automaton $\mathcal{A}$ which accepts a set C, we want to compute $pre^*(C)$ by constructing the automaton $\mathcal{A}_{pre^*}$.

Algorithm 1, shown in Figure 3.3, computes the transitions of $\mathcal{A}_{pre^*}$ by implementing the saturation rule from section 3.1.1. The idea of the algorithm is to avoid unnecessary operations. When we have a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$, we look out for pairs of transitions $t_1 = (p', \gamma', q')$ and $t_2 = (q', \gamma'', q'')$, where q', q'' are arbitrary states, so that we may add (p, γ, q'') – but we do not know the order in which such transitions will be added to the automaton. If every time we see a transition like t_2 we check for the existence of t_1, many checks might be negative and waste time to no avail. However, once we see t_1 we

Algorithm 1

Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$;
a $\mathcal{P}$-Automaton $\mathcal{A} = (\Gamma, Q, \rightarrow_0, P, F)$ without transitions into P

Output: the set of transitions of $\mathcal{A}_{pre^*}$

$rel := \emptyset$; $trans := \rightarrow_0$; $\Delta' := \emptyset$;
for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $trans := trans \cup \{(p, \gamma, p')\}$;
while $trans \neq \emptyset$ **do**
  pop $t = (q, \gamma, q')$ from $trans$;
  if $t \notin rel$ **then**
    $rel := rel \cup \{t\}$;
    for all $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in (\Delta \cup \Delta')$ **do**
      $trans := trans \cup \{(p_1, \gamma_1, q')\}$;
    for all $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta$ **do**
      $\Delta' := \Delta' \cup \{\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q', \gamma_2 \rangle\}$;
      for all $(q', \gamma_2, q'') \in rel$ **do**
        $trans := trans \cup \{(p_1, \gamma_1, q'')\}$;
return rel

Figure 3.3: An algorithm for computing pre^*.

know that all subsequent transitions (q', γ'', q'') must lead to (p, γ, q''). It so happens that the introduction of an extra rule $\langle p, \gamma \rangle \hookrightarrow \langle q', \gamma'' \rangle$ is enough to take care of exactly these cases. We collect these extra rules in a set called Δ'; this notation should make it clear that the pushdown system itself is not changed and that Δ' is merely needed for the computation. However, Δ' does play a useful rule when model-checking LTL (see Section 3.2.2).

The sets *rel* and *trans* contain transitions that are known to belong to $\mathcal{A}_{pre^*}$. The set *trans* contains those transitions which are waiting to be 'examined' – by this we mean that the algorithm still needs to check whether the presence of these transitions prompts further additions. When a transition is added to the automaton, it is first placed in *trans*. As long as *trans* is not empty, the algorithm picks one transition t from *trans*, moves it to *rel* and checks whether other transitions need to be added because of t.

Example: For a better illustration, consider how Algorithm 1 would process the example from Figure 3.1. The initialisation phase evaluates the ε-rules and adds (p_0, γ_1, p_0). When the latter is taken from *trans*, the rule

$\langle p_2, \gamma_2\rangle \hookrightarrow \langle p_0, \gamma_1\rangle$ is evaluated and (p_2, γ_2, p_0) is added. This, in combination with (p_0, γ_0, s_1) and the rule $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_0\rangle$, leads to (p_1, γ_1, s_1), and Δ' now contains a rule $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_0, \gamma_0\rangle$. We now have $p_1 \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_0} s_2$, so the next step adds (p_0, γ_0, s_2), and Δ' is extended by $\langle p_0, \gamma_0\rangle \hookrightarrow \langle s_1, \gamma_0\rangle$. Because of Δ', (p_0, γ_0, s_2) leads to the addition of (p_1, γ_1, s_2). Finally, Δ' is extended by $\langle p_0, \gamma_0\rangle \hookrightarrow \langle s_2, \gamma_0\rangle$, but no other transitions can be added and the algorithm terminates.

We now show that Algorithm 1 correctly implements the procedure from Section 3.1.1, the correctness of which has already been proved, and examine its complexity. The results are summarized in Theorem 3.4 at the end of this section. We need the following four lemmata:

Lemma 3.5 *For every transition t, if $t \in$ trans at any time during the execution of the algorithm, then t will be 'examined' exactly once.*

Proof: *rel* is initially empty, and every transition added to it is examined immediately afterwards. Until t is removed from *trans*, the algorithm cannot terminate. When t is removed from *trans*, it is examined if and only if it is not in *rel*, i.e. if and only if it has not been examined before, otherwise t is ignored. $\square$

Lemma 3.6 *Algorithm 1 terminates.*

Proof: *rel* is initially empty and can only grow afterwards. Q and Γ are finite sets, therefore *rel* can only contain finitely many elements. Thus, the part after line 6 can only be executed finitely many times. Because of this and because Δ is finite, there will be finitely many members of Δ', and so the loop at line 7 is traversed only finitely often. Thus, every execution of the part after line 6 terminates, and only finitely many elements can be added to *trans*. Once *rel* can no longer grow, *trans* can no longer grow either and will be empty eventually, which causes Algorithm 1 to terminate. $\square$

Lemma 3.7 *Upon termination of Algorithm 1 we have rel $= \rightarrow$, where $\rightarrow$ is the transition relation of $\mathcal{A}_{pre^*}$.*

Proof:

"$\subseteq$": We show that throughout the algorithm *rel* $\subseteq \rightarrow$ holds. In Section 3.1.1 $\rightarrow$ was defined to be the smallest relation containing $\rightarrow_0$ and satisfying the following:

If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$ and $p' \xrightarrow{w}^* q$, then $p \xrightarrow{\gamma} q$.

rel contains only elements from *trans*, so we inspect the lines that change *trans*. In line 1, *trans* initialized to $\rightarrow_0$. We show that all other additions are in compliance with the saturation rule:

- Lines 2 and 12 directly model the saturation rule.
- In line 8, if the rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle$ is taken from Δ, then we directly model the saturation rule. Otherwise, $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle$ was added to Δ' because $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p'', \gamma' \gamma \rangle \in \Delta$ and (p'', γ', q) in *trans* for some p'', γ' and again the saturation rule applies.

"$\supseteq$": We show that upon termination *rel* $\supseteq \rightarrow$ holds. After line 1, *trans* contains $\rightarrow_0$. Because of Lemma 3.5 all elements of *trans* will eventually end up in *rel*. Moreover, we prove that by the time the algorithm terminates, all possible applications of the saturation rule have been realized. Assume that we have $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and that there is a path from p' to q labelled by w in *rel*.

- If $w = \varepsilon$, then $q = p'$, and (p, γ, p') has been added in line 2.
- If $w = \gamma_1$, then there is some transition $t_1 = (p', \gamma_1, q)$ in *rel*. When t_1 was added to *rel*, (p, γ, q) was added to *trans* in line 8.
- If $w = \gamma_1 \gamma_2$, then *rel* contains $t_1 = (p', y_1, q')$ and $t_2 = (q', \gamma_2, q)$ for some state q'.
  - If t_1 was examined before t_2, Δ' has the rule $\langle p, \gamma \rangle \hookrightarrow \langle q', \gamma_2 \rangle$. Then, when t_2 was examined, (p, γ, q) was added in line 8.
  - If t_2 was examined before t_1, it was in *rel* when t_1 was examined. Then (p, γ, q) was added in line 12.

$\square$

Lemma 3.8 *Algorithm 1 takes* $O(|Q|^2 |\Delta|)$ *time and* $O(|Q|\,|\Delta| + |{\rightarrow_0}|)$ *space.*

Proof: Let $n_Q = |Q|$, $n_0 = |{\rightarrow_0}|$, and $n_\Delta = |\Delta|$. The size of $\mathcal{A}$ can be written as $n_Q + n_0$.

Imagine that prior to running the algorithm, all rules of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma' w\rangle$ have been put into 'buckets' labelled (p', γ'). If the buckets are organised in a hash table this can be done in $O(n_\Delta)$ time and space. Similarly, all rules in Δ' can be put into such buckets at run-time, and the addition of one rule takes only constant time.

If *rel* and $\rightarrow_0$ are implemented as hash tables, then addition and membership test take constant time. Moreover, if *trans* is a stack, then addition and removal of transitions (from the stack) take constant time, too. However, we will want to avoid adding any transition to *trans* more than once. This can be done (without asymptotic loss of time) by storing all transitions which are ever added to *trans* in an additional hash table. Transitions are added to the stack only if they are not already in the table. A more detailed discussion of useful data structures for this purpose can be found in Section 5.1.

When (q, γ, q') is examined, we need to regard just the rules in the bucket (q, γ). Because of Lemma 3.5, every possible transition is used at most once. Based on these observations, let us compute how often the statements inside the main loop are executed.

Line 10 is executed once for each combination of rules $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\gamma_2\rangle$ and transitions (q, γ, q'), i.e. $O(n_Q n_\Delta)$ times; hence the size of Δ' is $O(n_Q n_\Delta)$, too. For the loop starting at line 11, q' and γ_2 are fixed, so line 12 is executed $O(n_Q^2 n_\Delta)$ times.

Line 8 is executed once for each combination of rules $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$ in $(\Delta \cup \Delta')$ and transitions (q, γ, q'). As stated previously, the size of Δ' is $O(n_Q n_\Delta)$, so line 8 is executed $O(n_Q^2 n_\Delta)$ times.

Let us now count the iterations of the main loop, i.e. how often line 4 is executed. This directly depends on the number of elements that are added to *trans*. Initially, there are $n_0 + O(n_\Delta)$ elements from lines 1 and 2. Notice that $n_0 = O(n_Q \cdot n_\Delta \cdot n_Q)$. We already know that the other additions to *trans* are no more than $O(n_Q^2 n_\Delta)$ in number. As a conclusion, the whole algorithm takes $O(n_Q^2 n_\Delta)$ time.

Memory is needed for storing *rel*, *trans* and Δ'. The size of Δ' is $O(n_Q n_\Delta)$ (see above). There are n_0 additions to *trans* in line 1, and $O(n_\Delta)$ additions in line 2. In lines 8 and 12, p_1 and γ_1 are taken from the head of a rule in Δ. This means that these lines can only contribute $O(n_\Delta n_Q)$ different transitions.

From these facts it follows that the algorithm takes $O(n_Q n_\Delta + n_0)$ space, the size needed to store the result. Algorithm 1 is therefore optimal with respect to memory usage. □

Theorem 3.4 *Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system and let $\mathcal{A} = (\Gamma, Q, \rightarrow_0, P, F)$ be an automaton. There exists an automaton $\mathcal{A}_{pre^*}$ which recognizes $pre^*(L(\mathcal{A}))$. Moreover, $\mathcal{A}_{pre^*}$ can be constructed in $O(n_Q^2 n_\Delta)$ time and $O(n_Q n_\Delta + n_0)$ space, where $n_Q = |Q|$, $n_0 = |\rightarrow_0|$, and $n_\Delta = |\Delta|$.*

Proof: The existence of $\mathcal{A}_{pre^*}$ has been established by Theorem 3.2. According to Lemmata 3.6, 3.7, and 3.8, Algorithm 1 terminates, computes $\mathcal{A}_{pre^*}$, and requires no more than the time and space stated in the theorem. □

Conclusions The abstract procedure from Section 3.1.1 was first presented in [9]; its proof of correctness was also given there and has been repeated here for the sake of completeness. However, Theorem 3.4 provides an asymptotic improvement over the implementations given in [9], which were either $O(n_\mathcal{P}^2 n_\mathcal{A}^3)$ time and $O(n_\mathcal{P} n_\mathcal{A})$ space or $O(n_\mathcal{P} n_\mathcal{A}^2)$ time and $O(n_\mathcal{P} n_\mathcal{A}^2)$ space, where $n_\mathcal{P} = |P| + |\Delta|$ and $n_\mathcal{A} = |Q| + |\rightarrow_0|$.

It is easy to see that the bound still holds for pushdown systems in general (i.e. without the restriction of right-hand sides to length 2). All that is required is to apply Algorithm 1 to the pushdown system constructed by the method from Theorem 3.1, then remove from $\mathcal{A}_{pre^*}$ all transitions which carry additional stack symbols, which takes $\mathcal{O}(|\mathcal{A}_{pre^*}|)$ time.

3.1.4 An efficient implementation for *post**

Given a regular set of configurations C, we want to compute $post^*(C)$, i.e. the set of successors of C where C is represented by an automaton $\mathcal{A}$. Without loss of generality, we assume that $\mathcal{A}$ has no ε-transitions and no transitions leading to initial states.

Algorithm 2, shown in Figure 3.4, computes the transitions of $\mathcal{A}_{post^*}$ by implementing the saturation rule from Section 3.1.2. The approach is in some ways similar to the solution for pre^*; again we use *trans* and *rel* to store the transitions that we still need to examine or have already examined, respectively. Note that transitions from states outside of P go directly to *rel* since these states cannot occur in rules.

The algorithm is quite straightforward. We start by including the transitions of $\mathcal{A}$; then, for every transition that is known to belong to $\mathcal{A}_{post^*}$, we find its successors. A complication arises from the presence of ε-transitions, because paths of the form $p \xrightarrow{\gamma}{}^{*} q$ may involve more than one transition. This is resolved by combining transition pairs of the form $p \xrightarrow{\varepsilon} q'$ and $q' \xrightarrow{\gamma} q$ to $p \xrightarrow{\gamma} q$ whenever such a pair is detected.

Example: Consider again the example in Figure 3.2. The transitions $(p_1, \gamma_1, q_{p_1,\gamma_1})$ and $(q_{p_1,\gamma_1}, \gamma_0, s_1)$ are a consequence of (p_0, γ_0, s_1); the former gives the transitions $(p_2, \gamma_2, q_{p_2,\gamma_2})$ and $(q_{p_2,\gamma_2}, \gamma_0, q_{p_1,\gamma_1})$ and, in turn, $(p_0, \gamma_1, q_{p_2,\gamma_2})$. Because of $\langle p_0, \gamma_1\rangle \hookrightarrow \langle p_0, \varepsilon\rangle$, we add an ε-move from p_0 to q_{p_2,γ_2}. Having done this, we combine the ε-transition with all transitions that leave q_{p_2,γ_2}; in this example, we have $(q_{p_2,\gamma_2}, \gamma_0, q_{p_1,\gamma_1})$ and therefore we add $(p_0, \gamma_0, q_{p_1,\gamma_1})$. The latter finally leads to $(p_1, \gamma_1, q_{p_1,\gamma_1})$ and $(q_{p_1,\gamma_1}, \gamma_0, q_{p_1,\gamma_1})$. Figure 3.5 shows the result, similar to Figure 3.2 but with an additional transition from p_0 to q_{p_1,γ_1} which results from the combination of an ε-transition with another transition.

We now show that Algorithm 2 is a correct implementation of the procedure from Section 3.1.2, and also analyze its complexity. The results are summarized in Theorem 3.5 at the end of this section.

For a better understanding of the following paragraphs it is useful to consider the structure of the transitions in $\mathcal{A}_{post^*}$. Let $Q_1 = (Q \setminus P)$ and $Q_2 = (Q' \setminus Q)$. In the beginning, there are no transitions into P, i.e. we just have transitions from P into Q_1, and from Q_1 into Q_1. After line 15 is executed once, we also have transitions from P to Q_2. All the other additions to *rel* are now either from P to $Q_1 \cup Q_2$ except for line 16; here we have transitions from Q_2 to $Q_1 \cup Q_2$. We can summarise these observations in the following facts:

- After execution of the algorithm, *rel* contains no transitions leading into P.
- The algorithm does not add any transitions starting in Q_1.

Lemma 3.9 *Algorithm 2 terminates.*

Proof: Clearly, the size of Q' is still finite, so the maximum size of *rel* is $|Q'| \cdot |\Gamma + 1| \cdot |Q'|$ (accounting for ε-transitions), and we can use a similar reasoning as in Lemma 3.6, the corresponding lemma for *pre**. $\square$

Algorithm 2

Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$;
a $\mathcal{P}$-Automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ without transitions into P and without ε-transitions

Output: the automaton $\mathcal{A}_{post^*}$

$\mathit{trans} := (\rightarrow_0) \cap (P \times \Gamma \times Q)$;
$\mathit{rel} := (\rightarrow_0) \setminus \mathit{trans};\ Q' := Q$;
for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle \in \Delta$ **do**
  $Q' := Q' \cup \{q_{p',\gamma_1}\}$;
while $\mathit{trans} \neq \emptyset$ **do**
  pop $t = (p, \gamma, q)$ from trans;
  if $t \notin \mathit{rel}$ **then**
    $\mathit{rel} := \mathit{rel} \cup \{t\}$;
    if $\gamma \neq \varepsilon$ **then**
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \varepsilon, q)\}$;
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \rangle \in \Delta$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \gamma_1, q')\}$;
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle \in \Delta$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \gamma_1, q_{p',\gamma_1})\}$;
        $\mathit{rel} := \mathit{rel} \cup \{(q_{p',\gamma_1}, \gamma_2, q)\}$;
        for all $(p'', \varepsilon, q_{p',\gamma_1}) \in \mathit{rel}$ **do**
          $\mathit{trans} := \mathit{trans} \cup \{(p'', \gamma_2, q)\}$;
    else
      for all $(q, \gamma', q') \in \mathit{rel}$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p, \gamma', q')\}$;
return $(Q', \Gamma, \mathit{rel}, P, F)$

Figure 3.4: An algorithm for computing $post^*$.

Because Algorithm 2 combines ε-transitions with non-ε-transitions, we cannot show that $rel = \rightarrow$. Instead, we prove the following lemma which implies that the acceptance behaviour of the automaton computed in Algorithm 2 will be that of $\mathcal{A}_{post^*}$.

Lemma 3.10 *Let $\rightarrow$ be the transition relation of $\mathcal{A}_{post^*}$. Upon termination of the algorithm, $(q, \gamma', q') \in rel$ holds for any $q, q' \in Q'$ and $\gamma' \in \Gamma \cup \{\varepsilon\}$ if and only if $q \xrightarrow{\gamma'}{}^{*} q'$.*

Proof: Recall that in Section 3.1.2 $\rightarrow$ was defined to be the smallest relation containing $\rightarrow_0$, and satisfying the following:

- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ and $p \xrightarrow{\gamma}{}^{*} q$, then $p' \xrightarrow{\varepsilon} q$.
- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ and $p \xrightarrow{\gamma}{}^{*} q$, then $p' \xrightarrow{\gamma'} q$.
- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$ and $p \xrightarrow{\gamma}{}^{*} q$, then $p' \xrightarrow{\gamma'} q_{p',\gamma'} \xrightarrow{\gamma''} q$.

"$\Rightarrow$" We show that if $(q, \gamma', q') \in rel$, then $q \xrightarrow{\gamma'}{}^{*} q'$. Since elements from *trans* flow into *rel*, we inspect all the lines that change *trans* or *rel*:

- Lines 1 and 2 add elements from $\rightarrow_0$ which is a subset of $\rightarrow$.
- Line 11 is a case of the first part of the saturation rule.
- Line 13 is a case of the second part of the saturation rule.
- Lines 15 and 16 are a case of the third part of the saturation rule.

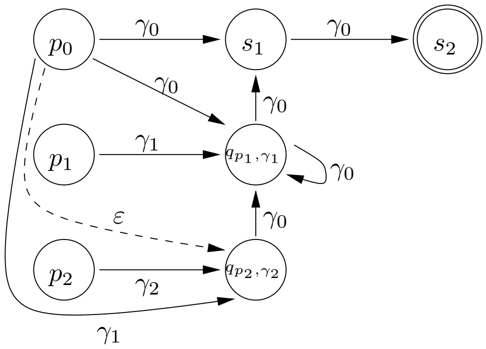

Figure 3.5: $\mathcal{A}_{post^*}$ as computed by Algorithm 2.

– In line 18 we have $(p'', \varepsilon, q_{p',\gamma_1})$ and $(q_{p',\gamma_1}, \gamma_2, q)$. Since both transitions must have resulted from some application of the saturation rule, we conclude that $p'' \xrightarrow{\gamma_2}{}^* q$ holds, so the addition of (p'', γ_2, q) is justified.

– Likewise, in line 21 we combine (p, ε, q) and (q, γ', q') to (p, γ', q').

"$\Leftarrow$" We show that if $q \xrightarrow{\gamma'}{}^* q'$, then after termination $(q, \gamma', q') \in rel$. By the same argumentation as in Lemma 3.5 we can say that all the elements in *trans* eventually end up in *rel*. Therefore it is sufficient to prove that all elements of $\rightarrow$ and all combinations involving ε-transitions are added to either *rel* or *trans* during execution of the algorithm.

We observe the following: Since there are no transitions leading into P, the ε-transitions can only go from states in P to states in $Q_1 \cup Q_2$, and no two ε-transitions can be adjacent. The relation $p \xrightarrow{\gamma}{}^* q$ can thus be written as follows:

$$p \xrightarrow{\gamma}{}^* q \iff p \xrightarrow{\gamma} q \vee \exists q' \colon p \xrightarrow{\varepsilon} q' \wedge q' \xrightarrow{\gamma} q$$

The desired property follows from the following facts:

– Because of lines 1 and 2, after execution $\rightarrow_0 \subseteq rel$ holds.

– If $\langle p', \gamma' \rangle \hookrightarrow \langle p, \varepsilon \rangle$ and $(p', \gamma', q) \in rel$, then (p, ε, q) is added in line 11.

– If $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma \rangle$ and $(p', \gamma', q) \in rel$, then (p, γ, q) is added in line 13.

– If $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ and $(p', \gamma', q) \in rel$, then $(p, \gamma, q_{p,\gamma})$ is added in line 15 and $(q_{p,\gamma}, \gamma'', q)$ in line 16.

– Whenever there is a pair (p, ε, q') and (q', γ, q) in *rel* for some $p, q', q \in Q'$ and $y \in \Gamma$, we need to add (p, γ, q).

  ∗ Assume that (q', γ, q) is known before (p, ε, q') is known. Then (p, γ, q) is added in line 21;

  ∗ Otherwise (p, ε, q') is known before (q', γ, q). Recall that $q' \in Q_1 \cup Q_2$. ε-transitions are added only after the initialization phase, and the only transitions starting in $Q_1 \cup Q_2$ which are added after initialization are those in line 16. In this case (p, γ, q) is added in line 18.

□

Lemma 3.11 *Algorithm 2 takes* $O(|P| \cdot |\Delta| \cdot (n_1 + n_2) + |P| \cdot |{\rightarrow_0}|)$ *time and space, 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* $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ *in* Δ.

Proof: Let $n_P, n_1, n_2, n_\Delta, n_0$ be the sizes of P, Q_1, Q_2, Δ and δ, respectively. Once again let *rel* and $\rightarrow_0$ be implemented as a hash table and *trans* as a stack (without duplicates), so that all the needed addition, membership test and removal operations take constant time.

The rules in Δ can be sorted into buckets according to their left-hand side at the cost of $O(\Delta)$ time and space, i.e. every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ is placed into the bucket labelled (p, γ). The transitions in *rel* can be sorted into buckets according to their source state (i.e. a transition (q, γ, q') would be sorted into a bucket labelled q). Since no transition is added to *rel* more than once, this costs no more than $O(|rel|)$ time and space.

For every transition $t = (p, \gamma, q) \in rel$, the part between lines 8 and 21 is executed only once. Because we just need go through the (p, γ)-bucket for rules when we examine t, we can state the following:

- Line 11 is executed once for every combination of rules $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and transitions (p, γ, q) of which there are at most $|\Delta|\,|Q_1 \cup Q_2|$ many, i.e. there are $O(n_\Delta(n_1 + n_2))$ many executions.
- Likewise, lines 13, 15, and 16 are also executed $O(n_\Delta(n_1 + n_2))$ times.
- Since there are at most n_P many transitions of the form (p, ε, q) for any given $q \in Q'$, line 18 is executed $O(n_P n_\Delta(n_1 + n_2))$ times.
- For line 21, we distinguish two cases:
  - $q \in Q_1$: Altogether, there are $O(n_0)$ many transitions going out from the states in Q_1, and each of them can be copied at most once to every state in P, which means $O(n_P n_0)$ many operations (remember that the algorithm adds no transitions which leave states in Q_1).
  - $q \in Q_2$: If $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle \in \Delta$, then all the transitions which leave q_{p_2,γ_2} are of the form $(q_{p_2,\gamma_2}, \gamma_3, q'')$ where q'' may be in $Q_1 \cup Q_2$. Therefore we end up with $O(n_P n_\Delta(n_1 + n_2))$ many operations.

- Line 6 is executed at most once for every transition added to *trans* as discussed above, i.e. $O(n_P n_\Delta(n_1+n_2)+n_P n_0)$ times. This is also an upper bound for the size of *rel* and *trans*.

- The initialisation phase can be completed in $O(n_0+n_\Delta+n_Q)$ (for the transitions in δ we just need to decide whether the source state is in P and add the transition to either *trans* or *rel*).

- Q' has $O(n_P+n_1+n_2)$ members.

$\square$

Theorem 3.5 *Let $\mathcal{P}=(P,\Gamma,\Delta,c_0)$ be a pushdown system, and let $\mathcal{A}=(Q,\Gamma,\rightarrow_0,P,F)$ be an automaton. There exists an automaton $\mathcal{A}_{post^*}$ which recognizes $post^*(L(\mathcal{A}))$. $\mathcal{A}_{post^*}$ can be constructed in $O(n_P n_\Delta(n_1+n_2)+n_P n_0)$ time and space, where $n_P=|P|$, $n_\Delta=|\Delta|$, $n_0=|\rightarrow_0|$, $n_1=|Q\setminus P|$, and n_2 is the number of different pairs (p,γ) such that there is a rule of the form $\langle p',\gamma'\rangle \hookrightarrow \langle p,\gamma\gamma''\rangle$ in Δ.*

Proof: The existence of $\mathcal{A}_{post^*}$ has been established by Theorem 3.3. According to Lemma 3.9, Algorithm 2 terminates. According to Lemma 3.10, the automaton constructed by the algorithm accepts $post^*(L(\mathcal{A}))$. Finally, according Lemma 3.11, the algorithm requires no more than the time and space stated in the theorem. $\square$

Conclusions In [22] the same problem was considered (with different restrictions on the rules in the pushdown system). The complexity of the *post** computation for the initial configuration was given as $O(n_{\mathcal{P}}^3)$ where $n_{\mathcal{P}}$ translates to n_P+n_Δ. An extension to compute *post**(C) for arbitrary regular sets C is also proposed. The different restrictions on the pushdown rules make a more detailed comparison difficult, but it is safe to say that Algorithm 2 is at least as good as the one in [22]. Also, we give an explicit bound for the computation of *post** for arbitrary regular sets of configurations. For a comparison with the bounds in [1] see Section 3.1.5.

3.1.5 The procedural case

Section 2.3 presented a translation from programs with procedures into pushdown systems. If we apply reachability analysis to this class of systems, we can interpret the actions of the algorithms in terms of the underlying programs. In the first part of this section, we give such an interpretation for *post**.

We proceed as follows: First, we embark on a fairly general discussion of four types of actions which any reachability algorithm for programs with procedures can be expected to perform. We then point out how the behaviour of *post** fits this description. Finally, we compare its behaviour to that of some algorithms for recursive state machines [1, 3, 6]. It turns out that there is a close relationship between all algorithms.

A general view Suppose that we are given a program with procedures, global, and local variables like the model discussed in Section 2.3, and that we are given the task of computing its set of reachable states. In the following, we make the assumption that reachability is computed in forward direction.

Let us define the head of a program state as its control point with an associated global and local valuation, but without any information about the rest of the call stack. We can distinguish two slightly different objectives: to compute the set of program states (complete with stack contents), or to compute the reachable heads only. In both cases, we need to take care of both actions within a procedure and interaction between procedures:

Intraprocedural propagation For propagation of reachability within a procedure, we shall use the concept of *path edges*. A path edge records the information that a certain head inside some procedure has been found to be reachable from some initial head of the same procedure (with whatever possible stack contents). Reachability within a procedure can be propagated by the following action:

1. If there is a path edge from initial an head x to some head y, and y can go to head z in of the same procedure, then a path edge from x to z must be added.

Interprocedural propagation To compute the interactions between procedures involves three more types of action:

2. When a procedure call is reached, the initial state of the callee must be marked reachable.

When a procedure terminates, we need to record the 'result' of the callee and feed it back into the caller. The result of the callee can be described by the values of the global variables upon termination. Since a callee can have multiple callers we need two actions for this:

3. When reaching a return statement, the algorithms need to resume reachability analysis at the return addresses of all corresponding call sites that have already proven to be reachable.

4. When reaching a call statement, the algorithms can check whether the effect of the callee has been computed previously and immediately continue reachability analysis at the return address.

The information needed by action 4 three can be prepared during action 3 in two different ways:

- *Lazy evaluation* records, for each procedure, a relation between initial heads and possible results. Subsequently, when a call is reached, this relation is used explicitly to produce a path edge.

- *Eager evaluation* inserts the result into every possible call site in the form of *summary edges* whenever results become known. A summary edge is essentially a new intraprocedural transition between the head where the call takes place and the corresponding return address.

Interpretation of post^* Assume that pushdown system $\mathcal{P}$ has the form $(G, \Gamma_0 \times L, \Delta, c_0)$ and that we want to compute $\mathit{post}^*(\{c_0\})$ where $c_0 = \langle g_0, (\mathit{main}_0, l_0)\rangle$ represents the initial state of a program with procedures, i.e. g_0 and l_0 are the initial states of the global and local variables.

We defined the head of a pushdown configuration $\langle g, (m, l)\, w\rangle$ as the pair $\langle g, (m, l)\rangle$. Thus, if a configuration represents a program state, the head of the configuration represents the head of the program state.

The automaton representing $\{c_0\}$ has the states $P \cup \{s\}$ (where s is a final state) and one edge $(g_0, (\mathit{main}_0, l_0), s)$. The post^* procedure first looks for rules of the form

$$\langle g, (n_1, l)\rangle \hookrightarrow \langle g', (m_0, l')\, (n_2, l'')\rangle$$

and adds a state $q_{g',(m_0,l')}$ for each of them. Notice that m_0 is the first control point in the callee function m and that g' and l' together make up the initial values of the variables and arguments passed to m. In other words, each new state $q_{g',(m_0,l')}$ corresponds to some initial head of a procedure m. In a similar vein, the state s can be seen as corresponding to the initial head of the procedure *main*.

By Lemma 3.4 we have that if $p \xrightarrow{w}{}^* q$ holds in the automaton $\mathcal{A}_{post^*}$ for some new state $q = q_{p,\gamma'}$, then $\langle p, \gamma' \rangle \Rightarrow^* \langle p, w \rangle$. In particular, this means that

$$g' \xrightarrow{(m',l')} q_{g,(m_0,l)} \qquad \Longleftrightarrow \qquad \langle g, (m_0, l) \rangle \Rightarrow^* \langle g', (m', l') \rangle$$

i.e. when entering the procedure at the head $\langle g, (m_0, l) \rangle$ one can reach the head $\langle g', (m', l') \rangle$ of the same procedure. Similarly, a transition $g' \xrightarrow{(m',l')} s$ means that $\langle g', (m', l') \rangle$ is a head of *main* which is reachable from the initial head. We conclude therefore that an automaton transition which originates from an initial state of $\mathcal{A}_{post^*}$ corresponds to a path edge in the sense defined earlier.

Intraprocedural progress (i.e. action 1) is computed by line 13 in Algorithm 2: As we have said, a transition $g' \xrightarrow{(m',l')} q_{g,(m_0,l)}$ corresponds to a path edge from $x = \langle m_0, (g, l) \rangle$ to $y = \langle m', (g', l') \rangle$. A step from y to a head $z = \langle m'', (g'', l'') \rangle$ is expressed by a rule $\langle g', (m', l') \rangle \hookrightarrow \langle g'', (m'', l'') \rangle$. If we have a rule and a transition of the above form, then line 13 extends the automaton by a transition $g'' \xrightarrow{(m'',l'')} q_{g,(m_0,l)}$, i.e. a path edge from x to z.

Some of the transitions produced by Algorithm 2 are labelled with ε. In light of Lemma 3.4 this means:

$$g \xrightarrow{\varepsilon} q_{g,(m,l)} \qquad \Longleftrightarrow \qquad \langle g, (m, l) \rangle \Rightarrow^* \langle g, \varepsilon \rangle$$

i.e. the ε-transitions are exactly the sort of relation needed for lazy evaluation. Suppose that we have a transition/path edge $g' \xrightarrow{(m',l')} q_{g,(m,l)}$ and a procedure call represented by a rule $\langle g', (m', l') \rangle \hookrightarrow \langle g_1, (m_1, l_1)\,(m'', l'') \rangle$. Then lines 15 and 16 add new transitions

$$g_1 \xrightarrow{(m_1,l_1)} q_{g_1,(m_1,l_1)} \xrightarrow{(m'',l'')} q_{g,(m,l)}$$

The first is a path edge from the initial head of the callee to itself; its addition corresponds to action 2. The other new transition has no counterpart in actions 1 to 4 and can be given the following interpretation: “The caller, starting at head $\langle g, (m, l) \rangle$, has invoked the callee at $\langle g_1, (m_1, l_1) \rangle$ and deposited a return address m'' and local variables l'' on the stack.” Given the previous arguments, it is relatively straightforward to see that line 18 corresponds to action 4, and line 21 to action 3.

Comparison with other algorithms The papers [1] and [6] propose recursive (also called hierarchical) state machines (RSMs) to model sequential programs with recursive procedure calls. The following definition is taken from [1] with minor differences in notation:

Definition 3.3 *A* recursive state machine *A is a tuple $(A_1, \ldots, A_k)$, where each component state machine $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$ consists of the following items:*

- *a set N_i of* nodes *and a disjoint set B_i of* boxes*;*
- *a labelling $Y_i\colon B_i \to \{1, \ldots, k\}$ that maps every box to the index of one of the component machines $A_1, \ldots, A_k$;*
- *a set of* entry nodes *$En_i \subseteq N_i$;*
- *a set of* exit nodes *$Ex_i \subseteq N_i$;*
- *a transition relation δ_i where transitions are of the form (u, v) where $u \in N_i$ or $u = (b, x)$ such that $b \in B_i$ and $x \in Ex_{Y_i(b)}$, and $v \in N_i$ or $v = (b, e)$ such that $b \in B_i$ and $e \in En_{Y_i(b)}$.*

Figure 3.6 shows an example which illustrates this definition. The example has three component state machines A_1, A_2, and A_3. Nodes are represented by circles, entry nodes on the left edge of their boxes and exit nodes on the right. A_1 contains a box which is mapped to A_2, A_2 contains a box mapped to A_3, and vice versa. The transition relation is indicated with arrows.

Each box can be said to represent a procedure. The node sets N_i represent the states of an execution of procedure i and encode both control flow and data values. An edge to a pair (b, e) in δ_i where $b \in B_i$ represents a call to

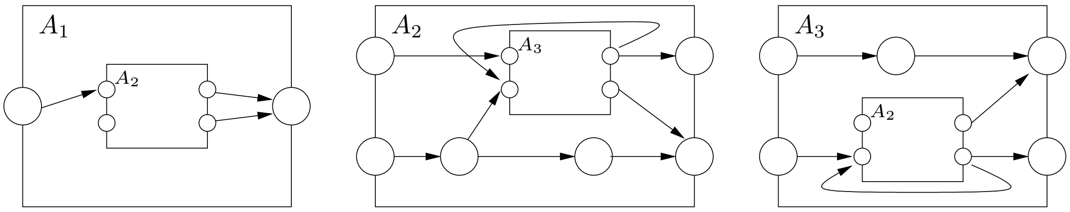

Figure 3.6: An example for a recursive state machine.

procedure $Y_i(b)$ such that the procedure is entered at entry node e. RSMs have the same expressive power as pushdown systems.

The Boolean Program (BP) model in [3] can be interpreted as a special case of RSM where control flow and data are encoded separately and where all variables, global and local, are of type Boolean. If control can pass from control point v to v', the admissible variable valuations for the transition are stored in a BDD $\mathit{Transfer}_{v,v'}$. The representation of BPs is therefore very similar to our notion of symbolic pushdown systems.

The algorithms proposed in [1, 3, 6] compute the heads of the reachable states, but not their associated stack contents. For the BP model, [3] presents a forward reachability algorithm; [1] and [6] propose very similar reachability algorithms for the RSM model which use forward analysis in procedures i where $|En_i| < |Ex_i|$ and backward analysis in the other procedures.

Path edges and intraprocedural propagation is handled analogously to the method described in action 1.

- The algorithms in [1, 6] (in their forward versions) maintain predicates of the form $R_i(x, y)$ where $x \in En_i$ and $y \in N_i$. The predicate $R_i(x, y)$ expresses that procedure i has a path edge from x to y. The valid predicates are determined as follows:

  If $R_i(x, y)$ and $\delta_i(y, z)$ hold, then so does $R_i(x, z)$.

  The backward versions are similar, but maintain path edges from the exit nodes to the inner nodes and propagate reachability backwards.

- The algorithm for the BP model [3] implements the idea in a symbolic setting: For each control point v, the BDD $\mathit{PathEdges}(v)$ is true at the element (g, l, g', l') if the valuation (g', l') (of the global and local variables) can be reached at control point v when entering the procedure with valuation (g, l). If v' is a successor of v, then $\mathit{PathEdges}(v')$ is extended by the join of $\mathit{PathEdges}(v)$ and $\mathit{Transfer}_{v,v'}$. Incidentally, in a symbolic setting the BDD operations used to implement this are the same as in Algorithm 2, see also Section 3.3.

Going into the details of how these algorithms handle actions 2,3, and 4 would lead too far at this point; we shall just point out that in [1] lazy evaluation is used whereas [3] and [6] use eager evaluation.

Conclusions To summarize the comparison we can say that despite the different models (RSM/BP vs pushdown systems) and representation issues (symbolic vs explicit) a substantial similarity is noticeable between the actions performed by all algorithms. All algorithms execute the four types of actions explained in the beginning of the section. The differences can be summarized as follows:

- Algorithm 2 computes all possible stack configurations whereas the others compute only the heads. The effort required to obtain this additional information is incurred by line 16 which has no counterpart in the other algorithms.

- We introduced the concepts of lazy and eager evaluation for analyzing interactions between procedures. Algorithm 2 and [1] use lazy evaluation, whereas [3] and [6] use eager evaluation. Section 5.6 examines the effect of this difference through experiments. These suggest (perhaps not very surprisingly) that lazy evaluation is faster.

- A 'philosophical' difference is that the RSM/BP model has a strong notion of procedures which the pushdown model has not. However, this does not seem to have an influence on efficiency or applicability of these models in practice.

- Algorithm 2 and the one in [3] use forward computation whereas [1] and [6] use a mix of forward and backward computation. While the comparison above focuses on their forward 'mode', a similar relationship could be established between pre^* (Algorithm 1) and their backward mode (which we have left out of the discussion here).

- Algorithm 2 and [3] have been implemented and tested on large examples (see Chapter 4), which the other two have not.

The fact that [1] and [6] have no implementations is regrettable, because their mixture of forward and backward search affords them a better asymptotic complexity, which would make a comparison on practical examples interesting. Their algorithms need $\mathcal{O}(\theta^2\,|\delta|)$ time where $\delta = \bigcup_{i=1}^{k} \delta_i$ and $\theta = \max_{1\le i\le k} \min(|En_i|, |Ex_i|)$.

When we translate an RSM to an equivalent pushdown system of the form $(G, \Gamma_0 \times L, \Delta, c_0)$, we get $|\Delta| = O(|\delta|)$, $|G| = \max_{1\le i\le k} |Ex_i|$, and $n_2 =$

$\sum_{i=1}^{k} |En_i|$, where n_2 is the number of newly added states. The estimation given by Theorem 3.5, applied to the type of pushdown system used in this subsection, is $\mathcal{O}(|G|\,|\Delta|\,n_2)$ time and space, which would make the estimation for [1] and [6] look far better. However, using Theorem 3.5 can be somewhat misleading here because the theorem does not make any assumptions at all about structural aspects of the pushdown system (which would be likely to improve the estimation for this special case). As we have seen, Algorithm 2 and the forward mode of [1] and [6] actually perform almost the same actions.

Thus, the main question would be what the mixture of forward and backward computation can achieve. This is difficult to predict because, even in cases where backward reachability has a better complexity estimation, it may perform worse in practice. This is discussed to a greater extent in Section 5.5.

3.1.6 Witness generation

Algorithms 1 and 2 compute, for a regular set of configurations C, the predecessors and successors of the elements of C. In practice it is often desirable to understand *how* a certain configuration was added to $pre^*(C)$ or $post^*(C)$, that is, given a configuration $c \in post^*(C)$ (resp. $c \in pre^*(C)$) we would like to find a configuration $c' \in C$ such that $c' \Rightarrow^* c$ (resp. $c \Rightarrow^* c'$) and reconstruct a 'witness' path between the two configurations. In the following we show how to obtain such witnesses with a simple extension of the existing algorithms and without affecting their complexity. We show the method for $post^*$; the corresponding method for pre^* is analogous.

To each transition t in $\mathcal{A}_{post^*}$ we add a label $\mathcal{L}(t)$ to record the 'reason' why the transition was added. Applying these reasons in reverse reconstructs the path by which each configuration was added. The method is first presented for the abstract algorithm from Section 3.1.2. Its extension to the implementation from Section 3.1.4 is then fairly straightforward.

Every transition t which is initially in $\mathcal{A}$ is given a special label $\perp$. All other transitions result from an application of the saturation rule which can be summarised as follows:

$$\begin{array}{c}\text{If } p \xrightarrow{\gamma}{}^{*}\, q \text{ and } r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta, \\ \text{add transitions to create a path } p' \xrightarrow{w}{}^{*}\, q.\end{array}$$

The 'reason' for the addition of $p' \xrightarrow{w}{}^{*} q$, as it were, is rule r and the presence of $p \xrightarrow{\gamma}{}^{*} q$. Labelling the transitions of $p' \xrightarrow{w}{}^{*} q$ with r already allows to

transition	*label*
(p_0, γ_0, s_1)	$\bot$
(s_1, γ_0, s_2)	$\bot$
$(p_1, \gamma_1, q_{p_1,\gamma_1})$	$(r_1, \bullet)$
$(q_{p_1,\gamma_1}, \gamma_0, s_1)$	$(r_1, \bullet)$
$(p_2, \gamma_2, q_{p_2,\gamma_2})$	$(r_2, \bullet)$
$(q_{p_2,\gamma_2}, \gamma_0, q_{p_1,\gamma_1})$	$(r_2, \bullet)$
$(p_0, \gamma_1, q_{p_2,\gamma_2})$	$(r_3, \bullet)$
$(p_0, \varepsilon, q_{p_2,\gamma_2})$	$(r_4, \bullet)$
$(q_{p_1,\gamma_1}, \gamma_0, q_{p_1,\gamma_1})$	(r_1, q_{p_2,γ_2})

$$
\begin{array}{l}
p_1 \xrightarrow{\gamma_1} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2 \\
p_0 \xrightarrow{\varepsilon} q_{p_2,\gamma_2} \xrightarrow{\gamma_0} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2 \\
p_0 \xrightarrow{\gamma_1} q_{p_2,\gamma_2} \xrightarrow{\gamma_0} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2 \\
p_2 \xrightarrow{\gamma_2} q_{p_2,\gamma_2} \xrightarrow{\gamma_0} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2 \\
p_1 \xrightarrow{\gamma_1} q_{p_1,\gamma_1} \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2 \\
p_0 \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_0} s_2
\end{array}
$$

Figure 3.7: Labels and reconstruction for $\mathcal{A}_{post^*}$ from Figure 3.2.

recover r, p, γ and q. Earlier, we noted that

$$p \xrightarrow{\gamma}{}^* q \iff p \xrightarrow{\gamma} q \vee \exists q' \colon p \xrightarrow{\varepsilon} q' \wedge q' \xrightarrow{\gamma} q.$$

Therefore we extend the label with q' if an ε-transition is involved and with $\bullet$ to indicate the other case. To summarize, each added transition gets the following label:

- if a transition is already present, the label is not changed;
- if $p \xrightarrow{\gamma} q$, set $\mathcal{L}(t) := (r, \bullet)$ for every new transition t;
- if $p \xrightarrow{\varepsilon} q' \xrightarrow{\gamma} q$, set $\mathcal{L}(t) := (r, q')$ for every new transition t.

Example The left half of Figure 3.7 shows the labels which would be computed in the example from Section 3.1.2.

Now, given a configuration $c = \langle p', w' \rangle \in post^*(C)$, the information contained in $\mathcal{L}$ will allow us to reconstruct a sequence from some $c' \in C$ to c in reverse order:

1. Find an accepting path $p' \xrightarrow{w'}{}^* q_f$ within $\mathcal{A}_{post^*}$.
2. Let $t_1 = (p', \gamma', q)$ be the first transition in the path (where $\gamma' \in \Gamma \cup \{\varepsilon\}$). If $\mathcal{L}(t_1) = \bot$, we have a configuration from C and are finished. Otherwise, let $\mathcal{L}(t_1) = (r, x)$.

3.1. If r has the form $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ with $|w| \leq 1$, obtain a new path by replacing t_1 with the transition (p, γ, q) if $x = \bullet$ and with (p, ε, x), (x, γ, q) otherwise.

3.2. Otherwise, r has a right-hand side of length 2. Let $t_2 = (q, \gamma'', q')$ be the second transition in the path and $\mathcal{L}(t_2) = (r', x')$; then r' has the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$. Obtain a new path by removing t_1 and t_2 and replacing them with (p, γ, q') if $x' = \bullet$ and with (p, ε, x'), (x', γ, q') otherwise.

4. The configuration represented by the new path is an immediate predecessor of the previous configuration. Go to step 2 for the next iteration.

Example (continued) The right half of Figure 3.7 illustrates the reconstruction procedure. It uses the labelling displayed on the left half of the figure to reconstruct a path from $\langle p_1, \gamma_1\gamma_0\gamma_0\gamma_0\rangle$ back to $\langle p_0, \gamma_0\gamma_0\rangle$, the only configuration accepted by the initial automaton.

The following two lemmata show that the reconstruction procedure is correct in the sense it indeed produces a valid path and that it ends at a configuration of C.

Lemma 3.12 *Given a configuration $c \in post^*(C)$, Steps 3.1 and 3.2 produce a predecessor of c which is also in $post^*(C)$.*

Proof: It is easy to see that Step 3.1 produces a direct predecessor of the previous configuration; only the prefix is changed and we just apply the rule r in reverse, at the same time obtaining a new path through the automaton. Step 3.2 follows the same principle but requires additional reasoning: If r has a right-hand side of length 2, then q can only be the state $q_{p',\gamma'}$, which was added by the *post** computation and which is not a final state. Hence the path *must* have a second transition. Inspecting the saturation rule and its augmentation above, we notice that all transitions of the form $(q_{p',\gamma'}, \gamma'', q')$ are added because of a path $p \xrightarrow{\gamma}{}^* q'$ and a rule $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$. Hence the reconstruction is correct in this case, too. (Notice that $\mathcal{L}(t_1)$ and $\mathcal{L}(t_2)$ may be different!) $\square$

Lemma 3.13 *The reconstruction procedure (Steps 1 to 4) terminates and ends at a configuration $c' \in C$.*

Proof: To prove that the procedure terminates we use a monotonicity argument. Let $age(t) = 0$ for every transition in $\mathcal{A}$, and let $age(t) = i$ if t is the i-th transition added to $\mathcal{A}_{post^*}$ during the construction. Let n be the number of transitions in $\mathcal{A}_{post^*}$ (note that n is finite). Let us associate a weight with each sequence of transitions:

$$\begin{aligned} weight(p \xrightarrow{\gamma} q) &:= age((p,\gamma,q))/n \\ weight(p \xrightarrow{w}{}^+ q' \xrightarrow{\gamma} q) &:= weight(p \xrightarrow{w}{}^+ q')/n + weight(q' \xrightarrow{\gamma} q) \end{aligned}$$

We observe that in step 3, the age of the transition(s) replacing t_1 is strictly smaller than $age(t_1)$. In step 4, the age of the transition(s) replacing t_1 and t_2 is strictly smaller than $age(t_2)$. This has two consequences:

1. When starting at a configuration $\langle p', w' \rangle$, the reconstruction procedure can never create a configuration with more than $|w'|+1+n$ transitions. This is because whenever the length of the configuration is extended, the age of the first transition decreases.

2. Given the observations above, it is easy to see that the weight strictly decreases with each iteration of the reconstruction procedure.

From 1. it follows that starting from a given configuration there are only finitely many different configurations which the reconstruction process can produce. From 2. it follows that the procedure cannot enter an infinite loop. Therefore, it must eventually satisfy the termination condition.

We still need to prove that $\mathcal{L}(t_1) = \bot$ in Step 2 implies that all transitions of the path (not only t_1) are from $\mathcal{A}$. This is easy because $\mathcal{L}(t_1) = \bot$ implies that q is a non-initial state. Since the algorithm does not add any transitions from non-initial states and since no transition lead to initial states, all transitions after t_1 must also be from $\mathcal{A}$. $\square$

The time and space complexity of the *post** algorithm is unchanged because the information added to each transition is of constant size and can be constructed in constant time.

When transferring these ideas to Algorithm 2, the only difference is that pairs of transitions $t_1 = (p, \varepsilon, q')$ and $t_2 = (q', \gamma, q)$ are combined to $t = (p, \gamma, q)$. This is easy enough to handle; we just mark t with q' to indicate that t results from such a combination. The rest of the procedure transfers in a straightforward way.

3.1.7 Shortest traces

Section 3.1.6 shows how to augment the *post** computation to obtain a path from a given $c \in post^*(C)$ back to some $c \in C$. In practice, one might be interested in obtaining the *shortest* path from c to c'. The reconstruction procedure itself is not concerned with the length of the sequence which it generates. Without further assumptions about what happens during the execution of *post**, it could actually produce a sequence which is exponentially long in the size of the pushdown system even when c could reach c' in just one step.

In the following we propose (and solve) a generalization of the shortest-path problem with non-negative edge lengths. The problem of finding the *length* of the shortest path from C to any configuration in $post^*(C)$ is a special case of this more general problem, and its solution can be combined with the method from Section 3.1.6 to reconstruct shortest paths. (As in Section 3.1.6 we present the solution for *post**, but an analogous method is available for *pre**.)

The problem is stated as follows:

- Let $\sqsubseteq$ be a total ordering on a domain D of *labels*. Let $\langle D, \oplus \rangle$ be a monoid whose neutral element $\bot$ is also the minimum w.r.t. $\sqsubseteq$. The operator $\oplus$ shall have the property that for any $a, b, c \in D$:

$$a \sqsubseteq b \quad \Longrightarrow \quad (a \oplus c \sqsubseteq b \oplus c) \wedge (c \oplus a \sqsubseteq c \oplus b) \tag{1}$$

  To facilitate the presentation we assume the existence of an "unreachable" element $\top$ which is the maximum of D w.r.t. $\sqsubseteq$ and has the property that $a \oplus b = \top$ implies $a = \top$ or $b = \top$. For technical convenience we define $\min_{\sqsubseteq} \emptyset := \top$. We write $a \sqsubset b$ if $a \sqsubseteq b$ and $a \neq b$.

- Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system and $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ be a $\mathcal{P}$-automaton which accepts C. Without loss of generality, we assume that $\mathcal{A}$ has no transitions to initial states and no ε-transitions.

- Let $f_\Delta \colon \Delta \rightarrow (D \setminus \{\top\})$ be a function which assigns a 'length' to each rule. Let $c \xRightarrow{\langle r \rangle} c'$ denote that c' can be reached from c by 'firing' r:

$$\langle p, \gamma w' \rangle \xRightarrow{\langle r \rangle} \langle p', w w' \rangle \qquad \Longleftrightarrow \qquad r = \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta,\ w' \in \Gamma^*$$

  Let $c \underset{d}{\Rightarrow}^{\oplus} c'$ denote the fact that there exists $n \geq 0$, configurations $c_1, \ldots, c_{n-1} \in Conf(\mathcal{P})$ and rules $r_1, \ldots, r_n \in \Delta$ such that

$$c \xRightarrow{\langle r_1 \rangle} c_1 \xRightarrow{\langle r_2 \rangle} \cdots \xRightarrow{\langle r_n \rangle} c' \quad \text{and} \quad f_\Delta(r_1) \oplus \cdots \oplus f_\Delta(r_n) = d$$

We have $c \overset{\oplus}{\underset{\bot}{\Longrightarrow}} c$ for all c.

- Let $S \subseteq \mathit{Conf}(\mathcal{P})$ be some set of configurations. $\|\cdot\|_S$ is defined (w.r.t. $\langle D, \oplus \rangle$, $\sqsubseteq$, and f_Δ) as a function $\|\cdot\|_S \colon \mathit{Conf}(\mathcal{P}) \to D$ with

$$\|c\|_S := \min\nolimits_{\sqsubseteq}\{\, d \mid \exists c' \in S,\ c' \overset{\oplus}{\underset{d}{\Longrightarrow}} c \,\}.$$

The problem consists of computing $\|\cdot\|_C$.

Intuitively we want to compute for each configuration c' the minimal length of a path from any $c \in C$ to c' where the length of a path is measured by the length of the rules fired along the way. In the following, we show that this can be achieved by imposing a discipline on the order in which Algorithm 2 processes the transitions it generates.

As usual, let $\mathcal{A}_{post^*} = (Q', \Gamma, \to, P, F)$ be the automaton which accepts $post^*(C)$. Let $\mathcal{L}\colon (Q' \times (\Gamma \cup \{\varepsilon\}) \times Q') \to D$ be a labelling of transitions in $\mathcal{A}_{post^*}$. For $q, q' \in Q'$ and $w \in \Gamma^*$ let $q \xrightarrow[d]{w}{}^{\oplus} q'$ denote the existence of states $q_1, \ldots, q_{n-1} \in Q'$ and $\gamma_1, \ldots, \gamma_n \in (\Gamma \cup \{\varepsilon\})$ such that the following holds:

$$\mathcal{L}(q_{n-1}, \gamma_n, q') \oplus \cdots \oplus \mathcal{L}(q, \gamma_1, q_1) = d \quad \text{and} \quad \gamma_1 \ldots \gamma_n = w$$

For any $q \in Q'$, $q \xrightarrow[\bot]{\varepsilon}{}^{\oplus} q$. Moreover, we denote minimal paths as follows:

$$|\langle p, w \rangle| := \min\nolimits_{\sqsubseteq}\{\, d \mid \exists q_f \in F \colon p \xrightarrow[d]{w}{}^{\oplus} q_f \,\}$$

The intuition for $\mathcal{L}$ is that the length of a path from C to some configuration c can be computed by adding the labels of the automaton transitions by which c is accepted in $\mathcal{A}_{post^*}$. We use $q \xrightarrow[d]{w}{}^{\oplus}_{i} q'$ and $|\langle p, w \rangle|_i$ to denote that the respective properties hold after i rounds of the modified $post^*$ algorithm which is outlined below:

Algorithm 3

1. Add new states as in the usual $post^*$ algorithm and set $\mathcal{L}(t) := \bot$ if $t \in \to_0$ and $\mathcal{L}(t) := \top$ otherwise. Set $i := 1$. Each round consists of the following steps:

2. Consider the priority of $(p, \gamma, q) \in P \times \Gamma \times (Q' \setminus P)$ as the minimal sum of the labels in an accepting path in the current automaton starting with $p \xrightarrow{\gamma}{}^{*} q$, i.e.

   $$\mathit{prio}_i(p, \gamma, q) := \min\nolimits_{\sqsubseteq}\{\, d_2 \oplus d_1 \mid p \xrightarrow[d_1]{\gamma}{}^{\oplus}_{i-1} q \xrightarrow[d_2]{w}{}^{\oplus}_{i-1} q_f,\ q_f \in F,\ w \in \Gamma^* \,\}$$

Choose a triple $(p_i^*, \gamma_i^*, q_i^*) \in P \times \Gamma \times (Q' \setminus P)$ with minimal priority among those triples that have not already been chosen in an earlier round.

3. Let $m_i^* := prio_i(p_i^*, \gamma_i^*, q_i^*)$ and let $b_i^* := \min_{\sqsubseteq}\{\, d \mid p_i^* \xrightarrow[d]{\gamma_i^*}{}^{\oplus}_{i-1} q_i^* \,\}$. If $m_i^* = \top$, terminate. Otherwise, for every rule of the form $r = \langle p_i^*, \gamma_i^* \rangle \hookrightarrow \langle p', w \rangle$ do the following:

   (i) If $|w| \le 1$, add a transition $t_1 = (p', w, q_i^*)$ and set $\mathcal{L}(t_1) := \min_{\sqsubseteq}\{\mathcal{L}(t_1), b_i^* \oplus f_\Delta(r)\}$.

   (ii) If $w = \gamma'\gamma''$, then add a transition $t_1 = (p', \gamma', q_{p',\gamma'})$ and set $\mathcal{L}(t_1) := \perp$, and add another transition $t_2 = (q_{p',\gamma'}, \gamma'', q_i^*)$ and set $\mathcal{L}(t_2) := \min_{\sqsubseteq}\{\mathcal{L}(t_2), b_i^* \oplus f_\Delta(r)\}$.

4. Set $i := i + 1$ and go back to Step 2.

It is easy to see that this extension still computes the same transitions as the procedure from Section 3.1.2 – we merely place a condition upon the order in which new transitions are processed. The termination condition expresses the fact that all matches for the saturation rule have been exhausted. Let n be the number of rounds it takes until the algorithm terminates. We prove two auxiliary lemmata:

Lemma 3.14 *For $0 \le i \le n$, $p \in P$, $w \in \Gamma^*$, $d \sqsubset \top$, if $p \xrightarrow[d]{w}{}^{\oplus}_{i} q$, then the following holds:*

(a) if $q \in F$, then $\langle p', w' \rangle \xRightarrow[d]{}{}^{\oplus} \langle p, w \rangle$ for a configuration $\langle p', w' \rangle \in C$;

(b) if $q = q_{p',\gamma'}$ is a new state, then $\langle p', \gamma' \rangle \xRightarrow[d]{}{}^{\oplus} \langle p, w \rangle$.

Proof: We proceed by induction over i.

Basis. $i = 0$.

(a) Then $\langle p, w \rangle \in C$ and $d = \perp$. Take $p' = p$ and $w' = w$.

(b) Then the left side is impossible, so we have nothing to show.

Step. $i > 0$. If $i > n$, we have nothing to prove. Otherwise, let j be the number of transitions occurring on $p \xrightarrow{w}{}^{*} q$ whose label was changed in the i-th round. We use induction on j. If $j = 0$, then $p \xrightarrow[d]{w}{}^{\oplus}_{i-1} q$ and we apply the induction hypothesis on i. So assume that $j > 0$ and

$$p \xrightarrow[d_1]{w_1}{}^{\oplus}_{i} q_1 \xrightarrow[d_2]{\gamma}{}^{\oplus}_{i} q_2 \xrightarrow[d_3]{w_2}{}^{\oplus}_{i-1} q \tag{2}$$

where $w_1\gamma w_2 = w$, $\gamma \in (\Gamma \cup \{\varepsilon\})$, $d_3 \oplus d_2 \oplus d_1 = d$, and (q_1, γ, q_2) is a transition whose label was changed in round i, and $p \xrightarrow{w_1}{}^{*} q_1$ has $j-1$ such transitions. We distinguish three cases:

1. If the label of $(q_1, \gamma, q_2) =: t_1$ was changed by case (i) of Step 3, then $w_1 = \varepsilon$, $d_1 = \bot$, $q_1 = p$. We conclude that there exists $r = \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p, \gamma \rangle \in \Delta$ such that

$$p_2 \xrightarrow[b_i^*]{\gamma_2}{}^{\oplus}_{i-1} q_2 \tag{3}$$

   Since $\mathcal{L}(t_1)$ was changed, $d_2 = b_i^* \oplus f_\Delta(r)$, so $p \xrightarrow[d_3 \oplus b_i^* \oplus f_\Delta(r)]{w}{}^{\oplus}_{i} q$.

   (a) From (2) and (3) and the induction hypothesis on i we get

$$\langle p', w' \rangle \xRightarrow[d_3 \oplus b_i^*]{}{}^{\oplus} \langle p_2, \gamma_2 w_2 \rangle$$

   for some $\langle p', w' \rangle \in C$, therefore

$$\langle p', w' \rangle \xRightarrow[d_3 \oplus b_i^* \oplus f_\Delta(r)]{}{}^{\oplus} \langle p, \gamma w_2 \rangle = \langle p, w \rangle,$$

   so the claim holds.

   (b) We argue along the same lines as in (a), substituting $\langle p', \gamma' \rangle$ for $\langle p', w' \rangle$.

2. If $(q_1, \gamma, q_2) =: t_1$ is the first of the transitions added by case (ii), then again $w_1 = \varepsilon$, $d_1 = \bot$, $q_1 = p$, and moreover $q_2 = q_{p,\gamma}$. We observe that $\mathcal{L}(t_1)$ is changed only once and from $\top$ to $\bot$, and when this happens in round i, no transitions are leaving q_2 before round i. Therefore $w_2 = \varepsilon$, $d_3 = \bot$, $q = q_2$, and for (a) there is nothing to prove. For (b), we simply have $\langle p', \gamma' \rangle = \langle p, \gamma \rangle \xRightarrow[\bot]{}{}^{\oplus} \langle p, \gamma \rangle = \langle p, w \rangle$, and since $d_2 = \bot$ the claim holds.

3. If $(q_1, \gamma, q_2) =: t_2$ is the second of the transitions added or modified in case (ii), let $r = \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p'', \gamma''\gamma \rangle$ be the responsible rule, so $q_1 = q_{p'',\gamma''}$ and

$$p_2 \xrightarrow[b_i^*]{\gamma_2}{}^{\oplus}_{i-1} \; q_2. \tag{4}$$

Since $\mathcal{L}(t_2)$ was changed, $d_2 = b_i^* \oplus f_\Delta(r)$. hence

$$p \xrightarrow[d_3 \oplus b_i^* \oplus f_\Delta(r) \oplus d_1]{w}{}^{\oplus}_{i} \; q.$$

Moreover, by the induction hypothesis on j, $\langle p, w_1 \rangle \overset{\oplus}{\underset{d_1}{\Longrightarrow}} \langle p'', \gamma'' \rangle$ holds.

   (a) Combining (2) and (4) with the induction hypothesis on i, we get

$$\begin{aligned} \langle p', w' \rangle &\overset{\oplus}{\underset{d_3 \oplus b_i^*}{\Longrightarrow}} \langle p_2, \gamma_2 w_2 \rangle \overset{\oplus}{\underset{f_\Delta(r)}{\Longrightarrow}} \langle p'', \gamma''\gamma w_2 \rangle \\ &\overset{\oplus}{\underset{d_1}{\Longrightarrow}} \langle p, w_1 \gamma w_2 \rangle = \langle p, w \rangle \end{aligned}$$

   for some $\langle p', w' \rangle \in C$. Therefore the claim holds.

   (b) Analogous to (a) with $\langle p', w' \rangle = \langle p', \gamma' \rangle$ if $q = q_{p',\gamma'}$.

□

Lemma 3.15 *For* $1 \le i \le n$ *and* $\langle p, w \rangle \in \mathit{Conf}(\mathcal{P})$ *the following holds:*

$$\|\langle p, w \rangle\|_C \sqsubset m_i^* \quad \Longrightarrow \quad |\langle p, w \rangle|_{i-1} = \|\langle p, w \rangle\|_C$$

Proof: $\|\langle p, w \rangle\|_C \sqsubset m_i^*$ implies $\|\langle p, w \rangle\|_C \sqsubset \top$, which means that in $\mathcal{T}_\mathcal{P}$ there exists a path of length k from some $c \in C$ to $\langle p, w \rangle$ such that $c \overset{\oplus}{\underset{\|\langle p,w \rangle\|_C}{\Longrightarrow}} \langle p, w \rangle$. We proceed by induction on k.

Basis. $k = 0$. Then $\langle p, w \rangle \in C$ and $|\langle p, w \rangle|_0 = \bot$ holds. Since the path by which $\langle p, w \rangle$ is accepted consists only of transitions whose label is $\bot$ initially, these labels will not change and so $|\langle p, w \rangle|_i = \bot = \|\langle p, w \rangle\|_C$ holds for $1 \le i \le n$.

Step. $k > 0$. Then there exists a configuration c' reachable in $k-1$ steps from some $c \in C$ and a rule r such that

$$c \overset{\oplus}{\underset{d}{\Longrightarrow}} c' \xRightarrow{\langle r \rangle} \langle p, w \rangle$$

and moreover $\|\langle p, w\rangle\|_C = d \oplus f_\Delta(r)$. Since $\|c'\|_C \sqsubseteq d \sqsubseteq d \oplus f_\Delta(r) \sqsubset m_i^*$ we can apply the induction hypothesis on k and obtain

$$|c'|_{i-1} = \|c'\|_C \tag{5}$$

In $d \sqsubseteq d \oplus f_\Delta(r)$, and in a couple of other places in the following, we make use of the property (1). Let $r = \langle p', \gamma' \rangle \hookrightarrow \langle p, w'' \rangle$, $c' = \langle p', \gamma' w' \rangle$, and $w = w''w'$ for suitable values of p', γ', w', w''. Because of (5), we have

$$p' \xrightarrow[d_1]{\gamma}{}^{\oplus}_{i-1} q' \xrightarrow[d_2]{w'}{}^{\oplus}_{i-1} q_f$$

for some $q' \in Q'$, $q_f \in F$, and d_1, d_2 with the property $d_2 \oplus d_1 = \|c'\|_C$. Therefore $prio_i(p', \gamma, q') \sqsubseteq \|c'\|_C \sqsubset m_i^*$, so (p', γ, q') must have been chosen in a round before i at which time a path $p \xrightarrow[d']{w''}{}^{\oplus}_{i-1} q'$, $d' \sqsubseteq d_1 \oplus f_\Delta(r)$ must have been added. Hence, we have

$$\begin{aligned} |\langle p, w\rangle|_{i-1} &\sqsubseteq d_2 \oplus d' \sqsubseteq d_2 \oplus d_1 \oplus f_\Delta(r) \\ &\sqsubseteq \|c'\|_C \oplus f_\Delta(r) \sqsubseteq d \oplus f_\Delta(r) = \|\langle p, w \rangle\|_C. \end{aligned}$$

Observe that Lemma 3.14 implies $\|\langle p, w\rangle\|_C \sqsubseteq |\langle p, w \rangle|_{i-1}$. Therefore we have $|\langle p, w\rangle|_{i-1} = \|\langle p, w\rangle\|_C$. □

Lemma 3.16 *Algorithm 3 effectively computes* $\|c\|_C$.

Proof: We claim that $|c| = \|c\|_C$ for all $c \in Conf(\mathcal{P})$: If c is reachable from some configuration in C, then $\|c\|_C \sqsubset \top$. Since $m_n^* = \top$, we have (according to Lemma 3.15) $|c|_n = |c| = \|c\|_C$. If c is not reachable from any configuration in C, we have $\|c\|_C = \top = |c|$ as a consequence of Lemma 3.14 (a). □

Computing priorities We now show how an implementation can compute the priorities efficiently. For this purpose we use a priority queue, i.e. a data structure which keeps a set of keys and associated values and provides efficient methods for adding key-value pairs and dequeueing the pair with the lowest value (see for instance [10]). Every triple (p, γ, q) is chosen only once. Therefore, we keep one pair in the queue for each triple which hasn't been chosen before and whose priority is not $\top$. The algorithm terminates when the priority queue is empty. Before round i, the value associated with (p, γ, q) in the queue shall be $prio_i(p, y, q)$. For efficiency reasons, we also keep keys (p, ε, q) in the queue whose value is the minimal sum of the labels in an accepting path starting with (p, ε, q). Moreover, we maintain a value $minpath(q)$ for every non-initial state $q \in (Q' \setminus P)$; its value shall be the minimal sum of the labels in a path starting in q and leading to a final state. We augment Algorithm 3 as follows:

Initially, for every transition $(p, \gamma, q') \in \rightarrow_0, \quad p \in P$ add the triple (p, γ, q') to the priority queue with value $\perp$. For every state $q \in (Q \setminus P)$ set $minpath(q) := \perp$, and for every state $q \in (Q' \setminus Q)$ set $minpath(q) := \top$.

In the following, when we say we 'update' a triple (p, γ, q) with a value d, we mean the following procedure: If $prio(p, \gamma, q) \sqsubseteq d$, we do nothing. If $prio(p, \gamma, q) = \top$, we add (p, γ, q) into the priority queue with value d. If $d \sqsubset prio(p, \gamma, q) \sqsubset \top$ (this will only happen when (p, γ, q) is already in the queue), we update the value of (p, γ, q) in the priority queue to d.

In round j of the algorithm, do the following: If the element dequeued from the priority queue is a triple (p, ε, q), then take all transitions of the form (q, γ, q') and update the triple (p, γ, q') with $minpath(q') \oplus \mathcal{L}(q, \gamma, q') \oplus \mathcal{L}(p, \varepsilon, q)$. We repeat this until the dequeued element is (p, γ, q) with $\gamma \in \Gamma$. Then we perform Step 3 with these additions:

- In case (i), update the triple (p, w, q) with $m_j^* \oplus f_\Delta(r)$ – notice that this is the shortest path created by the change of $\mathcal{L}(p, w, q)$.

- In case (ii), if $m_j^* \oplus f_\Delta(r) \sqsubset minpath(q_{p',\gamma'})$, set $minpath(q_{p',\gamma'}) := m_j^* \oplus f_\Delta(r)$ and update t_1 with the new value of $minpath(q_{p',\gamma'})$. This needs a bit of justification – why would t_1 be the only triple which needs updating? Let s be the old value of $minpath(q_{p',\gamma'})$. Then we had
$$m_j^* \sqsubseteq m_j^* \oplus f_\Delta(r) \sqsubset s \sqsubseteq prio_j(t_1)$$
Therefore, t_1 cannot have been chosen in round j or before, and the only path leading to $q_{p',\gamma'}$ consists of t_1, and hence it is enough to update only t_1.

- In case (ii), if $minpath(q_{p',\gamma'}) \sqsubseteq m_j^* \oplus f_\Delta(r)$ and $\mathcal{L}(t_2)$ was changed, go through all triples $(p'', \varepsilon, q_{p',\gamma'})$ which have already been chosen in an earlier round and update (p'', γ'', q) with $\mathcal{L}(p'', \varepsilon, q_{p',\gamma'}) \oplus \mathcal{L}(t_2)$. We need not update any non-ε-triples because we know that there are shorter paths leading away from $q_{p',\gamma'}$ than the one starting with t_2.

A comparison of this procedure with Algorithm 2 shows that the number of additions and updates to the priority queue is the same as the number of additions to *trans* and *rel* in Algorithm 2. Therefore, the complexity of the algorithm is increased by a factor κ which is the average time needed for an enqueue/dequeue operation on the priority queue (assuming that $\oplus$ operations take only constant time!). κ depends on certain properties of D and $\oplus$; for instance, if D has a constant, finite size, then these operations can be performed in constant time.

The following theorem summarizes the result of this section.

Theorem 3.6 *Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system, and let $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ be a $\mathcal{P}$-automaton. Let $\sqsubseteq$ be a total ordering on a monoid $\langle D, \oplus \rangle$ with property (1) and $f_\Delta \colon \Delta \rightarrow (D \setminus \{\top\})$ be a function. Then $\|\cdot\|_{L(\mathcal{A})}$ (w.r.t. $\langle D, \oplus \rangle$, $\sqsubseteq$, and f_Δ) can be effectively computed in*

$$O(\kappa \cdot (|P| \cdot |\Delta| \cdot (n_1 + n_2) + |P| \cdot |{\rightarrow_0}|))$$

time and

$$O(|P| \cdot |\Delta| \cdot (n_1 + n_2) + |P| \cdot |{\rightarrow_0}|)$$

space, where $n_1 = |Q \setminus P|$, n_2 is the number of different pairs (p, γ) with a rule of the form $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ in Δ, and κ is the average time for an enqueue/dequeue operation on a priority queue for values from D.

Conclusions Algorithm 3 solves a generalization of the shortest path problem with non-negative edge lengths. It also provides a unified solution for some problems proposed for pushdown systems and RSMs:

- For the Boolean Program setting (Sections 3.1.5 and 4.1.3) Ball and Rajamani [3] implemented a method which, given a reachable control point n in the program, constructs the shortest path from one particular initial configuration to any configuration involving n (but not for arbitrary stack configurations, or for arbitrary regular initial sets C).

- Jha and Reps [27] propose the use of reachability analysis on pushdown systems to solve a number of certificate-analysis problems in an authorization framework. Among other things, they suggest that automata transitions labelled by values from a lattice with meet and join operators could be used to answer questions like “How long is an authorization valid?” or “How trustworthy is an authorization?”. In the applications they suggest the lattices are totally ordered; therefore the algorithm above provides an efficient solution in those cases.

Practical issues of generating shortest traces are also discussed in Section 5.4.

3.1.8 Regular rewriting systems

The reachability algorithms presented in this section can be quite easily extended to the case where both sides of the rewriting rules are regular expressions.

Definition 3.4 *A* regular rewriting system *is a quadruple $\mathcal{R} = (P, \Gamma, \Delta, c_0)$ whose components P, Γ, and c_0 have the same role as in a pushdown system (w.l.o.g. we assume that c_0 has a non-empty stack). A rewrite rule $r \in \Delta$ has the form $((p, U_r), (p', V_r))$ such that $p, p' \in P$ and U_r, V_r are regular languages over Γ. With $\mathcal{R}$ we associate the transition system $\mathcal{T}_\mathcal{R} = (P \times \Gamma^*, \Rightarrow_\mathcal{R}, c_0)$ where $\Rightarrow_\mathcal{R}$ is the least relation satisfying the following:*

If $(\langle p, U\rangle, \langle p', V\rangle) \in \Delta,\;\; u \in U,\;\; v \in V,\;\; w \in \Gamma^*$, then $\langle p, uw\rangle \Rightarrow_\mathcal{R} \langle p', vw\rangle$.

The construction is quite straightforward: Given a regular rewriting system $\mathcal{R}$ we construct a pushdown system $\mathcal{P}_\mathcal{R}$ which simulates $\mathcal{R}$. We can then run the reachability algorithms on $\mathcal{P}_\mathcal{R}$ and remove additional configurations afterwards.

For every transition of $\mathcal{R}$, $\mathcal{P}_\mathcal{R}$ simulates finite automata for both sides of the corresponding rewrite rule. For each $r = ((p, U_r), (p', V_r)) \in \Delta$, let $\mathcal{U}_r = (S_r, \Gamma, \rightarrow_{\mathcal{U}_r}, s_r^0, F_r)$ and $\mathcal{V}_r = (T_r, \Gamma, \rightarrow_{\mathcal{V}_r}, t_r^0, G_r)$ be finite automata such that $L(\mathcal{U}_r) = \{\, w \in \Gamma^* \mid w^R \in U_r \,\}$ and $L(\mathcal{V}_r) = V_r$. Let $\bullet$ be a symbol not yet contained in Γ. Then $\mathcal{P}_\mathcal{R} = (P', \Gamma', \Delta', c_0)$ has the following components:

- $P' = P \cup \bigcup_{r\in\Delta}(S_r \cup T_r)$
- $\Gamma' = \Gamma \cup \{\bullet\}$
- For every $r = ((p, U_r), (p', V_r)) \in \Delta$, Δ' contains the following:

– if $s_r^0 \xrightarrow{\gamma}_{\mathcal{U}_r} s'$ then $\langle p, \gamma \rangle \hookrightarrow \langle s', \varepsilon \rangle$;

– if $s \xrightarrow{\gamma}_{\mathcal{U}_r} s'$ then $\langle s, \gamma \rangle \hookrightarrow \langle s', \varepsilon \rangle$;

– if $s \xrightarrow{\gamma}_{\mathcal{U}_r} s'$ and $s' \in F_r$ then $\langle s, \gamma \rangle \hookrightarrow \langle t_r^0, \bullet \rangle$;

– if $s_r^0 \xrightarrow{\gamma}_{\mathcal{U}_r} s'$ and $s' \in F_r$ then $\langle p, \gamma \rangle \hookrightarrow \langle t_r^0, \bullet \rangle$;

– if $s_r^0 \in F_r$ then $\langle p, \gamma \rangle \hookrightarrow \langle t_r^0, \bullet\, \gamma \rangle$;

– if $t \xrightarrow{\gamma}_{\mathcal{V}_r} t'$ then $\langle t, \bullet \rangle \hookrightarrow \langle t', \bullet\, \gamma \rangle$;

– if $t \in G_r$ then $\langle t, \bullet \rangle \hookrightarrow \langle p', \varepsilon \rangle$.

In other words, from any configuration of $\mathcal{R}$, $\mathcal{P}_{\mathcal{R}}$ can nondeterministically choose to execute any rule of $r \in \Delta$ without knowing whether the rule is applicable, i.e. whether the current stack content has a prefix from U_r. However, if the rule is not applicable, the system will be unable to reach another configuration from $\mathcal{R}$. Therefore the reachability algorithms applied to $\mathcal{P}_{\mathcal{R}}$ compute automata which accept exactly the set of configurations reachable in $\mathcal{R}$ plus additional configurations which start in $\bigcup_{r \in \Delta}(S_r \cup T_r)$. To remove these additional configurations it suffices to restrict the initial states of these automata to P.

A possible application for regular rewriting systems in verification concerns the modelling of Java-style exceptions. In this setting, the program may raise different types of exceptions which cause control to be transferred to the currently active 'handler' for the respective type of exception. Each block of the program can define such handlers for one or more types of exceptions. A handler becomes active when execution enters the block within which it is defined; when execution leaves the block, the previously active handler becomes reactivated. Therefore, whenever an exception is thrown, the Java engine has to walk the stack of activation records from top to bottom and find the first return address belonging to a block which has defined a handler for the exception. The latter property can be determined statically; if N is the set of program points, let N_e be the set of program points belonging to a block with a handler for an expression of type e, and for each handler h let N_h be the set of program points in the block to which h belongs. A program with exceptions can then be translated into a regular rewriting system along the lines of Section 2.3 with the following additions:

- each instruction which throws an exception of type e is translated to a rule which pushes the symbol t_e onto the stack;

- each handler h for exceptions of type e gives rise to a rule whose left-hand side regular language is $\{\, t_e w n \mid w \in (N - N_e)^*, n \in N_h \,\}$ and whose right-hand side language is $\{h_0\}$, where h_0 is the entry point of the handler.

An alternative modelling of Java-style exceptions in a pushdown framework is discussed by Obdržálek in [33].

Another class of transition systems studied in the literature are *prefix-recognizable rewrite systems* (PRRS) where, in addition to having regular languages on both sides of the rewrite rules, transitions are guarded by regular predicates on the rest of the stack content. The approach taken in this subsection can be extended to PRRS by combining it with methods similar to those of Chapter 6. A solution for model-checking the μ-calculus on PRRS is presented in [29].

3.2 Model-checking problems for LTL

In this section we present solutions to the problems (III) to (V) presented in Section 2.5, which concern model-checking question for pushdown systems and linear-time temporal logics (LTL). Our approach can be characterised as an adaptation of the automata-theoretic approach which was introduced by Vardi and Wolper [41] for finite-state systems.

The principal idea of the automata-theoretic approach is as follows: We are given a transition system $\mathcal{T} = (S, \rightarrow, s_0)$ and an LTL formula φ. We consider unlabelled transition systems here since we use a state-based interpretation of the logics given by a valuation $\nu\colon At(\varphi) \rightarrow S$. According to Theorem 2.2 there is a Büchi automaton $\mathcal{B} = (Q, 2^{At(\varphi)}, \delta, q_0, F)$ which accepts $L(\neg\varphi)$. (We denote the transition relation of $\mathcal{B}$ by the letter δ instead of $\rightarrow$ from here on to avoid it from being confused with transition relations of finite automata.) It is then possible to synchronize $\mathcal{T}$ and $\mathcal{B}$ to a new transition system $\mathcal{BT} = (S \times Q, \rightarrow', (s_0, q_0))$ in such a way that for each run σ of $\mathcal{BT}$ (with $\sigma(i) = (s_i, q_i)$ for $i \geq 0$) the following holds:

- the projection $\sigma^{\mathcal{T}}$ given by $\sigma^{\mathcal{T}}(i) = s_i$, $i \geq 0$, is a run of $\mathcal{T}$;

- the projection of σ to the states of $\mathcal{B}$, i.e. $\sigma^{\mathcal{B}}(i) = q_i$, $i \geq 0$, is a run of $\mathcal{B}$ labelled by $\sigma^{\mathcal{T}}_{\nu}$;

- $\mathcal{T} \not\models^{\nu} \varphi$ if and only if $\mathcal{BT}$ has a run σ starting at (s_0, q_0) such that $\sigma_{\nu}^{\mathcal{T}} \in L(\neg\varphi)$.

Determining whether the satisfaction condition holds is equivalent to checking whether $\mathcal{BT}$ has a run σ starting at (s_0, q_0) such that $\sigma^{\mathcal{B}} \cap \mathit{Inf}(F) \neq \emptyset$. The corresponding run $\sigma^{\mathcal{T}}$ satisfies $\neg\varphi$ (i.e. $s_0 \not\models^{\nu} \varphi$) and serves as a witness to the (negative) result. The construction of $\mathcal{BT}$ is as follows:

If $s \to s'$ and $(q, \mathcal{A}, q') \in \delta$ such that $\mathcal{A} = \{\, A \mid A \in At(\varphi),\ s \in \nu(A) \,\}$, then $(s, q) \to' (s', q')$.

We cannot transfer this approach to infinite-state systems completely without restrictions. First, we need the synchronized system to be finitely representable. Secondly, membership in $\nu(A)$ needs to be decidable in order to construct the synchronized system. A convenient and natural way to satisfy both constraints is to allow only *simple* valuations:

Definition 3.5 *Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system. A set of configurations of $\mathcal{P}$ is* simple *if it has the form $\{\, \langle p, \gamma w \rangle \mid w \in \Gamma^* \,\}$ for some $p \in P$, $\gamma \in \Gamma$. A valuation $\nu\colon At \to 2^{P \times \Gamma^*}$ is* simple *if $\nu(A)$ is a union of simple sets for every $A \in At$.*

A simple valuation can therefore describe a property of the control location and the topmost stack symbol of a configuration, but not of the rest of the stack. While this may seem quite restrictive at first, one can argue that for many applications the valuation can still 'talk' about the most important aspects of the configurations. For instance, in program analysis the control location contains the global variables and the topmost stack symbol the control location and local variables of the currently active procedure, if the translation from Section 2.3 is used. Also, simple valuations do not provide for the case where the stack is completely empty. However, this is not a real problem because we can always add an artificial 'bottom-of-stack' symbol to the system. For the rest of this chapter, we shall assume that all valuations are simple; in Chapter 6 we show how to lift the methods presented here to the case where valuations may include regular properties of the stack.

We adapt the automata-theoretic approach to pushdown systems by constructing Büchi pushdown systems:

Definition 3.6 *Let $\mathcal{P} = (P, \Gamma, \Delta, \langle p_0, w_0 \rangle)$ be a pushdown system, and let $\mathcal{B} = (Q, 2^{At(\varphi)}, \delta, q_0, F)$ be a Büchi automaton which accepts $L(\neg\varphi)$ for some LTL formula φ the validity of which is interpreted by a simple valuation ν. The product of $\mathcal{P}$ and $\mathcal{B}$ (with respect to ν) yields a* Büchi pushdown system $\mathcal{BP} = (P \times Q, \Gamma, \Delta', \langle (p_0, q_0), w_0 \rangle, G)$, *where*

- $\langle (p, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (p', q'), w \rangle$ *if* $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$ *and* $(q, \mathcal{A}, q') \in \delta$ *such that* $\mathcal{A} = \{\, A \mid A \in At(\varphi),\ \langle p, \gamma \rangle \in \nu(A) \,\}$*;*
- $(p, q) \in G$ *if* $q \in F$*.*

The transition system $\mathcal{T}_{\mathcal{BP}}$ associated with $\mathcal{BP}$ is equal to that of the pushdown system $((P \times Q), \Gamma, \Delta', \langle (p_0, q_0), w_0 \rangle)$.

Notice that $\mathcal{T}_{\mathcal{BP}}$ is a synchronization of $\mathcal{T}_{\mathcal{P}}$ and $\mathcal{B}$ in the same sense as explained earlier because the left-hand sides of the rules ensure that all configurations which can 'fire' them are compatible with the corresponding Büchi transition. In the following we assume that all Büchi pushdown systems are generated as above and denote their control locations by a simple set P, rather than $P \times Q$, to simplify the notation. In order to reason about these systems we use the following definitions:

Definition 3.7 *Let $\mathcal{BP} = (P, \Gamma, \Delta, c_0, G)$ be a Büchi pushdown system. An* accepting run *of $\mathcal{P}$ is a run which visits infinitely often configurations whose control locations are in G. The relation $\Rightarrow^r$ between configurations of $\mathcal{BP}$ is defined as follows: $c \Rightarrow^r c'$ if and only if $c \Rightarrow^* \langle g, w \rangle \Rightarrow^+ c'$ for some configuration $\langle g, w \rangle$ with $g \in G$. A head $\langle p, \gamma \rangle$ is* repeating *if there exists $v \in \Gamma^*$ such that $\langle p, \gamma \rangle \Rightarrow^r \langle p, \gamma v \rangle$. The set of repeating heads of $\mathcal{BP}$ is denoted by $Rep_{\mathcal{BP}}$ (we omit the index if $\mathcal{BP}$ is understood).*

If $\mathcal{BP}$ is the product of a pushdown system $\mathcal{P}$ and a Büchi automaton for $\neg\varphi$, then an accepting run σ starting from some $\langle (p, q_0), w \rangle$ exists if and only if the configuration $\langle p, w \rangle$ of $\mathcal{P}$ violates φ. The model-checking problems (III) to (V) thus boil down to detecting accepting runs in a Büchi pushdown system. The following proposition characterizes the configurations from which there are accepting runs.

Proposition 3.1 [9] *Let c be a configuration of a Büchi pushdown system $\mathcal{BP} = (P, \Gamma, \Delta, c_0, G)$. $\mathcal{BP}$ has an accepting run starting from c if and only if there exists a repeating head $\langle p, \gamma \rangle$ such that $c \Rightarrow^* \langle p, \gamma w \rangle$ for some $w \in \Gamma^*$.*

Proof: "$\Rightarrow$": Let σ be an accepting run where $\sigma(0) = c$ and where w_i is the stack content of $\sigma(i)$, $i \geq 0$. Construct a strictly increasing sequence of indices $i_0, i_1, \ldots$ with the following properties:

$$|w_{i_0}| = \min\{\, |w_j| \mid j \geq 0 \,\}$$
$$|w_{i_k}| = \min\{\, |w_j| \mid j > i_{k-1} \,\}, \quad k \geq 1$$

In other words, once any of the configurations $\sigma(i_k)$ is reached, the rest of the run never changes the bottom $|w_{i_k}| - 1$ elements of the stack anymore. Since $\mathcal{BP}$ has only finitely many different heads, there must be a pair $\langle p, \gamma \rangle$ which occurs infinitely often as a head in the sequence $\sigma(i_0)\sigma(i_1)\ldots$. Moreover, since some $g \in G$ becomes a control location infinitely often, we can find a subsequence of indices $i_{j_0}, i_{j_1}, \ldots$ with the following property: For every $k \geq 0$ there are $v, w \in \Gamma^*$ such that

$$\sigma(i_{j_k}) = \langle p, \gamma w \rangle \Rightarrow^r \langle p, \gamma v w \rangle = \sigma(i_{j_{k+1}})$$

In fact, because w is never looked at or changed in this path, we have $\langle p, \gamma \rangle \Rightarrow^r \langle p, \gamma v \rangle$ which proves this direction of the proposition.

"$\Leftarrow$": Since $\langle p, \gamma \rangle$ is a repeating head, we can construct the following run for some $u, v, w \in \Gamma^*$ and $g \in G$:

$$c \Rightarrow^* \langle p, \gamma w \rangle \Rightarrow^* \langle g, uw \rangle \Rightarrow^+ \langle p, \gamma v w \rangle \Rightarrow^* \langle g, uvw \rangle \Rightarrow^+ \langle p, \gamma v v w \rangle \Rightarrow^* \cdots$$

Since g occurs infinitely often, the run is accepting. $\square$

Subject to a method for finding the repeating heads, Proposition 3.1 gives us strategies for solving the model-checking problems. For the rest of the section we fix a pushdown system $\mathcal{P} = (P', \Gamma, \Delta, \langle p_0, w_0 \rangle)$ and an LTL formula φ whose negation is accepted by a Büchi automaton $\mathcal{B} = (Q, 2^{At(\varphi)}, \delta, q_0, F)$. We let $\mathcal{BP} = (P, \Gamma, \Delta, \langle (p_0, q_0), w_0 \rangle, G)$, where $P := P' \times Q$ and $G := P' \times F$, be the synchronization of $\mathcal{P}$ and $\mathcal{B}$. Moreover, we define $Rep\,\Gamma^*$ to denote $\{\, \langle p, \gamma w \rangle \mid \langle p, \gamma \rangle \in Rep,\ w \in \Gamma^* \,\}$.

(III) To find whether $\langle p_0, w_0 \rangle \models^\nu \varphi$, we check one of these two properties:

- $\langle (p_0, q_0), w_0 \rangle \in pre^*(Rep\,\Gamma^*)$
- $post^*(\{\langle (p_0, q_0), w_0 \rangle\}) \cap Rep\,\Gamma^* \neq \emptyset$

(IV) To solve the global model-checking problem, i.e. find the set of all configurations of $\mathcal{P}$ which violate φ, we compute the set

– $pre^*(Rep\,\Gamma^*) \cap \{\, \langle (p, q_0), w\rangle \mid p \in P',\ w \in \Gamma^* \,\}$

(V) To find all reachable configurations which violate φ, we compute the intersection between the solution to (II), i.e. $post^*(\{\langle p_0, w_0\rangle\})$, and (IV).

All the sets of configurations involved in these computations are regular, so we can use the methods from Section 3.1. In this section, we first present a solution to the problem of finding the set of repeating heads and then analyse the complexity of the model-checking procedures.

3.2.1 Computing the repeating heads

An important realization is that when we are looking for a path between $\langle p, \gamma\rangle$ and $\langle p, \gamma v\rangle$, we are not interested in the actual contents of v. We can thus determine the repeating heads based solely on the information about which *heads* are reachable from each other (while possibly pushing arbitrary symbols onto the stack) and whether an accepting state is passed in between. This information can be encoded in a finite graph; the problem of finding repeating heads in this graph then becomes quite similar to the problem of finding accepting cycles in a finite-state system.

Definition 3.8 *The* head reachability graph *of $\mathcal{BP}$ is the directed labelled graph $\mathcal{G} = (P \times \Gamma, \{0, 1\}, \rightarrow)$ whose nodes are the heads of $\mathcal{BP}$ and whose edges are labelled with* 0 *or* 1*. We have $(p, \gamma) \xrightarrow{b} (p', \gamma')$ under the following conditions:*

- *there exists a rule $\langle p, \gamma\rangle \hookrightarrow \langle p'', v_1\gamma' v_2\rangle$ for some $v_1, v_2 \in \Gamma^*$, $p'' \in P$;*
- *$\langle p'', v_1\rangle \Rightarrow^* \langle p', \varepsilon\rangle$;*
- *$b = 1$ if and only if $p \in G$ or $\langle p'', v_1\rangle \Rightarrow^r \langle p', \varepsilon\rangle$*

The instances for which $\langle p, v\rangle \Rightarrow^r \langle p', \varepsilon\rangle$ holds can be found with small modifications to the algorithms for pre^* or $post^*$. This is discussed in Section 3.2.2 for pre^*. Once $\mathcal{G}$ is constructed, *Rep* can be computed by exploiting the fact that some head $\langle p, \gamma\rangle$ is repeating if and only if (p, γ) is part of a strongly connected component of $\mathcal{G}$ which has a 1-labelled edge connecting two of its nodes. This is proved in Theorem 3.7 at the end of this section.

$P = \{p_0, p_1, p_2\}$
$\Gamma = \{\gamma_0, \gamma_1, \gamma_2\}$
$\Delta = \{r_1, r_2, r_3, r_4\}$

$r_1 = \langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1\gamma_0 \rangle$
$r_2 = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_0 \rangle$
$r_3 = \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle$
$r_4 = \langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle$

$G = \{p_2\}$

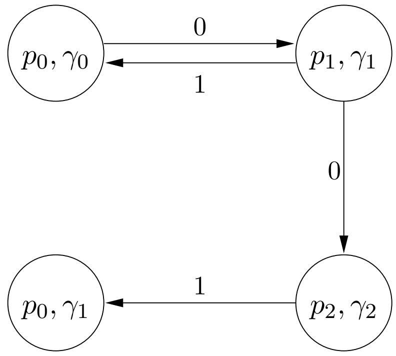

Figure 3.8: The graph $\mathcal{G}$.

Example Consider a Büchi pushdown system where P, Δ, Γ, and G are as shown in the left half of Figure 3.8. The right half of the figure shows the graph $\mathcal{G}$. Three of the edges are immediately derived from r_1, r_2, and r_3; the edge from $\langle p_1, \gamma_1 \rangle$ to $\langle p_0, \gamma_0 \rangle$ exists because

$$\langle p_1, \gamma_1 \rangle \Rightarrow \langle p_2, \gamma_2\gamma_0 \rangle \Rightarrow \langle p_0, \gamma_1\gamma_0 \rangle \Rightarrow \langle p_0, \gamma_0 \rangle;$$

it is labelled with 1 because $p_2 \in G$. The SCCs of $\mathcal{G}$ are $\{(p_0, \gamma_0), (p_1, \gamma_1)\}$, $\{(p_0, \gamma_1)\}$, and $\{(p_2, \gamma_2)\}$. Only the first of these has an internal 1-edge, so $\langle p_0, \gamma_0 \rangle$ and $\langle p_1, \gamma_1 \rangle$ are the repeating heads.

In the following we write $(p, \gamma) \rightarrow^r (p', \gamma')$ to express that there exist heads (p_1, γ_1) and (p_2, γ_2) such that

$$(p, \gamma) \rightarrow^* (p_1, \gamma_1) \xrightarrow{1} (p_2, \gamma_2) \rightarrow^* (p', \gamma')$$

Lemma 3.17 *The relations $\rightarrow^*$ and $\rightarrow^r$ have the following properties:*

(a) *$(p, \gamma) \rightarrow^* (p', \gamma')$ if and only if $\langle p, \gamma \rangle \Rightarrow^* \langle p', \gamma' v \rangle$ for some $v \in \Gamma^*$.*

(b) *Moreover, $(p, \gamma) \rightarrow^r (p', \gamma')$ if and only if $\langle p, \gamma \rangle \Rightarrow^r \langle p', \gamma' v \rangle$.*

Proof:
"$\Rightarrow$": Assume $(p, \gamma) \xrightarrow[i]{} (p', \gamma')$. We proceed by induction on i.

(a) **Basis.** $i = 0$. In this case, $(p, \gamma) = (p', \gamma')$, thus $\langle p, \gamma \rangle \Rightarrow^* \langle p', \gamma' \rangle$.

Step. $i > 0$. Then there exist p'', γ'' such that $(p, \gamma) \rightarrow (p'', \gamma'')$ and $(p'', \gamma'') \xrightarrow[i-1]{} (p', \gamma')$. From the induction hypothesis, there exists u such that $\langle p'', \gamma'' \rangle \Rightarrow^* \langle p', \gamma' u \rangle$. Since $(p, \gamma) \rightarrow (p'', \gamma'')$ we have $\langle p, \gamma \rangle \Rightarrow^* \langle p'', \gamma'' w \rangle$ (for some $w \in \Gamma^*$), hence $\langle p, \gamma \rangle \Rightarrow^* \langle p', \gamma' uw \rangle$.

(b) **Basis.** $i = 0$. In this case $(p, \gamma) \rightarrow^r (p', \gamma')$ cannot hold.

Step. $i > 0$. In the proof of (a) we had $(p, \gamma) \rightarrow (p'', \gamma'') \xrightarrow[i-1]{} (p', \gamma')$. In order for $(p, \gamma) \rightarrow^r (p', \gamma')$ to hold, either $(p'', \gamma'') \rightarrow^r (p', \gamma')$ holds or $(p, \gamma) \xrightarrow{1} (p'', \gamma'')$. In the first case, (by the induction hypothesis) $\langle p'', \gamma'' \rangle \Rightarrow^r \langle p', \gamma' u \rangle$ holds and hence $\langle p, \gamma \rangle \Rightarrow^r \langle p', \gamma' uw \rangle$.

In the second case $(p, \gamma) \xrightarrow{1} (p'', \gamma'')$ implies that there exists a rule $\langle p, \gamma \rangle \hookrightarrow \langle p_1, v_1 \gamma'' w \rangle$, and either $p \in G$ or $\langle p_1, v_1 \rangle \Rightarrow^r \langle p'', \varepsilon \rangle$ holds. If $p \in G$, we have $\langle p, \gamma \rangle \Rightarrow^r \langle p_1, v_1 \gamma'' w \rangle$. Otherwise $\langle p_1, v_1 \gamma'' w \rangle \Rightarrow^r \langle p'', \gamma'' w \rangle$ holds. By (a) we have $\langle p'', \gamma'' \rangle \Rightarrow^* \langle p', \gamma' u \rangle$. Again, in both cases we have $\langle p, \gamma \rangle \Rightarrow^r \langle p', \gamma' uw \rangle$.

"$\Leftarrow$": Assume $\langle p, \gamma \rangle \underset{i}{\Rightarrow} \langle p', \gamma' v \rangle$. We proceed by induction on i.

(a) **Basis.** $i = 0$. In this case we have $\langle p, \gamma \rangle = \langle p', \gamma' \rangle$ and $v = \varepsilon$, and $(p, \gamma) \rightarrow^* (p, \gamma)$ holds trivially.

Step. $i > 0$. Then there exists some configuration $\langle p'', u \rangle$ such that $\langle p, \gamma \rangle \underset{1}{\Rightarrow} \langle p'', u \rangle \underset{i-1}{\Longrightarrow} \langle p', \gamma' v \rangle$. There must be a rule $\langle p, \gamma \rangle \hookrightarrow \langle p'', u \rangle$, and moreover $|u| \geq 1$. Let l denote the minimal length of the stack on the path from $\langle p'', u \rangle$ to $\langle p', \gamma' v \rangle$. Then u can be written as $u'' \gamma_1 u'$ where $|u'| = l - 1$. Furthermore, there exists p''' such that $\langle p'', u'' \rangle \Rightarrow^* \langle p''', \varepsilon \rangle$ and, since u' has to remain on the stack for the rest of the path, v is of the form $v' u'$ for some $v' \in \Gamma^*$. This means that $\langle p''', \gamma_1 \rangle \underset{j}{\Rightarrow} \langle p', \gamma' v' \rangle$ for $j < i$. By the induction hypothesis, $(p''', \gamma_1) \rightarrow^* (p', \gamma')$. Also, we have $(p, \gamma) \rightarrow (p''', \gamma_1)$, hence $(p, \gamma) \rightarrow^* (p', \gamma')$.

(b) **Basis.** $i = 0$. $\langle p, \gamma \rangle \Rightarrow^r \langle p', \gamma' v \rangle$ is impossible in zero steps.

Step. $i > 0$. In the step of (a), either $\langle p, \gamma \rangle \Rightarrow^r \langle p'', u \rangle$ or $\langle p'', u \rangle \Rightarrow^r \langle p', \gamma' v \rangle$ holds. The first case implies $p \in G$. Then it holds that $(p, \gamma) \xrightarrow{1} (p''', \gamma_1)$ and $(p''', \gamma_1) \rightarrow^* (p', \gamma')$, thus $(p, \gamma) \rightarrow^r (p', \gamma')$.

In the second case $\langle p'', u \rangle \Rightarrow^r \langle p', \gamma' v \rangle$ holds. Either we have $\langle p'', u'' \rangle \Rightarrow^r \langle p''', \varepsilon \rangle$, then $(p, \gamma) \xrightarrow{1} (p''', \gamma_1)$, or $(p''', \gamma_1) \Rightarrow^r \langle p', \gamma' v' \rangle$, then by the induction hypothesis we have $(p''', \gamma_1) \rightarrow^r (p', \gamma')$. In both cases we get the desired result.

□

We can now prove the following theorem.

Theorem 3.7 *Let $\mathcal{BP}$ be a Büchi pushdown system, and let $\mathcal{G}$ be the associated head reachability graph. A head $\langle p, \gamma\rangle$ of $\mathcal{BP}$ is repeating if and only if the node (p, γ) is in a strongly connected component of $\mathcal{G}$ which has an internal* 1*-labelled edge (i.e. an edge connecting two nodes of the SCC).*

Proof: Let $\langle p, \gamma\rangle \in Rep$, i.e. there exists $v \in \Gamma^*$ such that $\langle p, \gamma\rangle \Rightarrow^r \langle p, \gamma v\rangle$. By Lemma 3.17, this is the case if and only if $(p, \gamma) \rightarrow^r (p, \gamma)$. From the definition of $\rightarrow^r$ it follows that there are (p', γ') and (p'', γ'') such that $(p, \gamma) \rightarrow^* (p', \gamma') \xrightarrow{1} (p'', \gamma'') \rightarrow^* (p, \gamma)$. Then (p, γ), (p', γ'), and (p'', γ'') are all in the same strongly connected component, and this component contains a 1-labelled edge. Conversely, whenever (p, γ) is in a component with such an edge, $(p, \gamma) \rightarrow^r (p, \gamma)$ holds. $\square$

Implementation To compute the repeating heads, we proceed in two phases. In the first phase, the cases for which $\langle p, w\rangle \Rightarrow^* \langle p', \varepsilon\rangle$ (resp. $\Rightarrow^r$) holds are computed, which allows us to construct $\mathcal{G}$. In the second phase, we identify the strongly connected components in $\mathcal{G}$. The latter is achieved by Tarjan's algorithm [40] which takes linear time in the size of $\mathcal{G}$.

Problem (III) can be solved by two methods, one involving *pre**, the other *post**. To avoid redundant computations, we can adapt the first phase to the method being used.

3.2.2 The *pre** method

Algorithm 4 in Figure 3.9 constructs $\mathcal{G}$ from information obtained while computing *pre** on the set $L_\varepsilon := \{\, \langle p, \varepsilon\rangle \mid p \in P \,\}$. Every resulting transition (p, γ, p') signifies that $\langle p, \gamma\rangle \Rightarrow^* \langle p', \varepsilon\rangle$ holds. However, we also need the information whether $\langle p, \gamma\rangle \Rightarrow^r \langle p', \varepsilon\rangle$. To this end, we enrich the alphabet of the automaton; instead of transitions of the form (p, γ, p') we now have transitions $(p, [\gamma, b], p')$ where b is a boolean. The meaning of a transition $(p, [\gamma, 1], p')$ should be that $\langle p, \gamma\rangle \Rightarrow^r \langle p', \varepsilon\rangle$. In the algorithm $G(p)$ yields 1 if and only if $p \in G$.

Termination: Termination follows from the fact that the *pre** algorithm terminates – we just have a larger alphabet.

Algorithm 4
Input: a Büchi pushdown system $\mathcal{BP} = (P, \Gamma, \Delta, c_0, G)$
Output: the edges of the graph $\mathcal{G} = (P \times \Gamma, \{0,1\}, \rightarrow)$

$\quad rel := \emptyset;\ trans := \emptyset;\ \Delta' := \emptyset;$
$\quad$ **for all** $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ **do**
$\qquad trans := trans \cup \{(p, [\gamma, G(p)], p')\};$
$\quad$ **while** $trans \neq \emptyset$ **do**
$\qquad$ pop $t = (p, [\gamma, b], p')$ from $trans$;
$\qquad$ **if** $t \notin rel$ **then**
$\qquad\quad rel := rel \cup \{t\};$
$\qquad\quad$ **for all** $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p, \gamma\rangle \in \Delta$ **do**
$\qquad\qquad trans := trans \cup \{(p_1, [\gamma_1, b \vee G(p_1)], p')\};$
$\qquad\quad$ **for all** $\langle p_1, \gamma_1\rangle \xhookrightarrow{b'} \langle p, \gamma\rangle \in \Delta'$ **do**
$\qquad\qquad trans := trans \cup \{(p_1, [\gamma_1, b \vee b'], p')\};$
$\qquad\quad$ **for all** $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p, \gamma\gamma_2\rangle \in \Delta$ **do**
$\qquad\qquad \Delta' := \Delta' \cup \{\langle p_1, \gamma_1\rangle \xhookrightarrow{b \vee G(p_1)} \langle p', \gamma_2\rangle\};$
$\qquad\qquad$ **for all** $(p', [\gamma_2, b'], p'') \in rel$ **do**
$\qquad\qquad\quad trans := trans \cup \{(p_1, [\gamma_1, b \vee b' \vee G(p_1)], p'')\};$

$\quad$ **for all** $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ **do add** $(p, \gamma) \xrightarrow{G(p)} (p', \gamma');$
$\quad$ **for all** $\langle p, \gamma\rangle \xhookrightarrow{b} \langle p', \gamma'\rangle \in \Delta'$ **do add** $(p, \gamma) \xrightarrow{b} (p', \gamma');$
$\quad$ **for all** $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ **do add** $(p, \gamma) \xrightarrow{G(p)} (p', \gamma');$

Figure 3.9: Computing the head reachability graph with the pre^* method.

Correctness: The first phase of the algorithm computes $pre^*(L_\varepsilon)$. The only modification to the algorithm for pre^* is the addition of the boolean indicators. We have already proved the correctness of the normal pre^* algorithm. Applying Lemmas 3.1 and 3.2 we get that $\langle p,\gamma\rangle \Rightarrow^* \langle p',\varepsilon\rangle$ if and only if a transition $(p,[\gamma,b],p')$ is produced for some b. It remains to show that a transition $(p,[\gamma,1],p')$ emerges if and only if $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$.

"$\Rightarrow$": We consider the circumstances under which 1-labelled transitions can be added to *trans*.

- Originally, the automaton contains no transitions at all.
- In line 3, we have $\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle$ and $p \in G$. Hence, $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$.
- In line 9, we have $\langle p_1,\gamma_1\rangle \hookrightarrow \langle p,\gamma\rangle$ and $(p,[\gamma,b],p')$. For $b \vee G(p_1)$ to be 1, either $b=1$ (which means $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$) or $p_1 \in G$ (which means $\langle p_1,\gamma_1\rangle \Rightarrow^r \langle p,\gamma\rangle$). In both cases we get $\langle p_1,\gamma_1\rangle \Rightarrow^r \langle p',\varepsilon\rangle$.
- In line 11, there is a transition $(p,[\gamma,b],p')$, and the existence of the rule means $\langle p_1,\gamma_1\rangle \hookrightarrow \langle p_2,\gamma_2\gamma\rangle$, $(p_2,[\gamma_2,b''],p)$, and $b' = G(p_1) \vee b''$ for some p_2,γ_2. In other words, $\langle p_1,\gamma_1\rangle \Rightarrow \langle p_2,\gamma_2\gamma\rangle \Rightarrow^* \langle p,\gamma\rangle \Rightarrow^* \langle p',\varepsilon\rangle$. If $b \vee b'$ is true, either $b=1$ (meaning $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$), or $p_1 \in G$ (meaning $\langle p_1,\gamma_1\rangle \Rightarrow^r \langle p_2,\gamma_2\gamma\rangle$), or $b''=1$ (meaning $\langle p_2,\gamma_2\gamma\rangle \Rightarrow^r \langle p,\gamma\rangle$).
- In line 15, the situation is similar to the preceding case, only the roles of p,γ and p_2,γ_2 are reversed.

"$\Leftarrow$": If $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$ holds, then by definition there exists $\langle g,u\rangle$ such that $\langle p,\gamma\rangle \xRightarrow[i_1]{} \langle g,u\rangle \xRightarrow[i_2]{} \langle p',\varepsilon\rangle$ where $i_1 \geq 0,\ i_2 \geq 1$. We prove the property by induction on $i := i_1 + i_2$.

Basis. $i=0$. By definition, $\langle p,\gamma\rangle \Rightarrow^r \langle p',\varepsilon\rangle$ cannot hold if $i=0$.

Step. $i>0$. Then we can find a configuration $\langle p'',w\rangle$ such that $\langle p,\gamma\rangle \xRightarrow[1]{} \langle p'',w\rangle \xRightarrow[i-1]{} \langle p',\varepsilon\rangle$ holds. Then $p'' \xrightarrow{w}{}^* p'$ in A_{pre^*}.

Either $p \in G$ so that $\langle p,\gamma\rangle \Rightarrow^r \langle p'',w\rangle$ holds. Then the addition rule of the pre^*-algorithm leads to $(p,[\gamma,1],p')$.

Otherwise $\langle p'',w\rangle \Rightarrow^r \langle p',\varepsilon\rangle$ holds. Then $|w| \geq 1$ since at least one step must occur. In the case $|w|=1$ let $w=\gamma_1$. By the induction hypothesis, $(p'',[\gamma_1,1],p')$ is in *rel*. Then $(p,[\gamma,1],p')$ will be added in line 9.

In the case $|w| = 2$ let $w = \gamma_1\gamma_2$. There exists p''' such that $\langle p'', \gamma_1\rangle \Rightarrow^* \langle p''', \varepsilon\rangle$ and $\langle p''', \gamma_2\rangle \Rightarrow^* \langle p', \varepsilon\rangle$. The length of both derivations will be at most $i - 1$, and we conclude that the algorithm must find transitions of the form $(p'', [\gamma_1, b_1], p''')$ and $(p''', [\gamma_2, b_2], p')$. Either $\langle p'', \gamma_1\rangle \Rightarrow^r \langle p''', \varepsilon\rangle$, then by the induction hypothesis $b_1 = 1$. Otherwise $\langle p''', \gamma_2\rangle \Rightarrow^r \langle p', \varepsilon\rangle$ and $b_2 = 1$. Depending on which transition is processed first by the algorithm, $(p, [\gamma, 1], p')$ is added in line 11 or 15.

Notice that we get $r' = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta'$ exactly if there is a rule $\langle p, \gamma\rangle \hookrightarrow \langle p'', \gamma''\gamma'\rangle \in \Delta$ and $(p'', [\gamma'', b], p') \in \mathit{rel}$ for some b which means $\langle p'', \gamma''\rangle \Rightarrow^* \langle p', \varepsilon\rangle$. Thus, by Definition 3.8 the existence of r' is equivalent to an edge in $\mathcal{G}$ whose label can be determined from b and p.

Complexity: Let $n_P = |P|$ and $n_\Delta = |\Delta|$. The first phase is essentially the same as Algorithm 1 with an initially empty automaton; the addition of boolean indicators on the arcs can only lead to a linear increase in both time and space requirements. Therefore, Theorem 3.4 tells us that the first phase takes $O(n_P^2 n_\Delta)$ time and $O(n_P n_\Delta)$ space.

The rules of Δ contribute $O(n_\Delta)$ nodes and edges to the size of $\mathcal{G}$. As the size of Δ' is $O(n_P n_\Delta)$, the total size of $\mathcal{G}$ is $O(n_P n_\Delta)$, too. Determining the strongly connected components is possible in linear time in the size of the graph [40] and the same holds for searching each component for an internal 1-edge.

Theorem 3.8 *Let $\mathcal{BP} = (P, \Gamma, \Delta, c_0, G)$ be a Büchi pushdown system. The set of repeating heads Rep can be computed in $O(n_P^2 n_\Delta)$ time and $O(n_P n_\Delta)$ space where $n_P = |P|$ and $n_\Delta = |\Delta|$.*

In the context of the model-checking problems presented earlier, we often need the repeating heads to compute $pre^*(\mathit{Rep}\,\Gamma^*)$. In this case we would need two rounds of pre^*; the first on L_ε (Algorithm 4), the second on $\mathit{Rep}\,\Gamma^*$. L_ε is represented by an automaton whose states and transitions are a subset of those used for $\mathit{Rep}\,\Gamma^*$. Therefore, the transitions obtained in the first round would also emerge in the second round (minus the boolean values). This, and the special form of the automaton for $\mathit{Rep}\,\Gamma^*$ mean that the second round can be very much simplified:

1. We start with the automaton obtained by Algorithm 4, but ignore all boolean values on its transitions and on Δ', and add a new state s which is the only non-initial and final state.

2. For each $\langle p, \gamma \rangle \in \mathit{Rep}$, add a transition (p, γ, s); for each $\gamma \in \Gamma$ add (s, γ, s).

3. Line 2 of Algorithm 1 has already been performed, so we leave it out. In line 4, t always has the form (p, γ, s). Lines 7 to 12 can be simplified as follows:

   for $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma \rangle \in (\Delta \cup \Delta')$ **do**
   $\quad \mathit{trans} := \mathit{trans} \cup \{(p_1, \gamma_1, s)\}$;
   for $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma\gamma_2 \rangle \in \Delta$ **do**
   $\quad \mathit{trans} := \mathit{trans} \cup \{(p_1, \gamma_1, s)\}$;

   In the second loop, we do not need to add rules to Δ'; they would have the form $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle s, \gamma_2 \rangle$ and could only lead to (p_1, γ_1, s) which we're adding anyway. Similarly, lines 11 and 12 from Algorithm 1 can be removed because they also lead to (p_1, γ_1, s).

4. The time and space taken up by Steps 2 and 3 is $\mathcal{O}(|\Delta| \cdot |Q|)$.

3.2.3 The *post** method

To solve the question whether the initial configuration $\langle p_0, w_0 \rangle$ satisfies an LTL formula φ, we can also check if $\mathit{post}^*(\{\langle (p_0, q_0), w_0 \rangle\}) \cap \mathit{Rep}\,\Gamma^* \neq \emptyset$. This question can be decided by the following procedure:

1. Compute the automaton $\mathcal{A}_{\mathit{post}^*}$ which represents $\mathit{post}^*(\{\langle (p_0, q_0), w_0 \rangle\})$.

2. It suffices to construct $\mathcal{G}$ restricted to the reachable heads. A head $\langle p, \gamma \rangle$ is reachable if there exists a transition (p, γ, q) (for any state q) in $\mathcal{A}_{\mathit{post}^*}$.

3. If the restriction of $\mathcal{G}$ to the reachable heads has a strongly connected component with a 1-labelled edge, then $\langle p_0, w_0 \rangle$ violates φ.

To construct $\mathcal{G}$, we again need the information if $\langle p, \gamma \rangle \Rightarrow^* \langle p', \varepsilon \rangle$ holds for certain $p, p' \in P,\ \gamma \in \Gamma$. According to 3.4, this is the case if in $\mathcal{A}_{\mathit{post}^*}$ we have an ε-transition between p' and $q_{p,\gamma}$. To see if $\langle p, \gamma \rangle \Rightarrow^r \langle p', \varepsilon \rangle$ we can again extend the transition with a boolean value. This is not presented in detail here as the principle is very similar to that of Algorithm 4; the idea is to annotate a transition $p \xrightarrow{\gamma} q_{p,\gamma}$ with 0 and update that value on subsequent transitions to $q_{p,\gamma}$ depending on which rules were used to add them.

3.2.4 Summary of model-checking problems

Using the earlier results from this section we can now compute the complexity of the problems (III) to (V). The following steps are necessary to solve the model-checking problems for a given pushdown system $\mathcal{P} = (P, \Gamma, \Delta, \langle p_0, w_0 \rangle)$ and a formula φ:

- Construct the Büchi automaton $\mathcal{B} = (Q, 2^{At(\varphi)}, \delta, q_0, F)$ corresponding to the negation of φ.

- Compute the Büchi pushdown system $\mathcal{BP}$ as the product of $\mathcal{B}$ and $\mathcal{P}$. Let $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, and $n_\delta = |\delta|$. Then $\mathcal{BP}$ has at most $s := n_P n_Q$ control locations and $r := n_\Delta n_\delta$ rules and can be computed in $\mathcal{O}(r)$ time.

In the *pre** method (see Section 3.2.2) we proceed as follows:

- Compute the set of repeating heads *Rep* of $\mathcal{BP}$. According to Theorem 3.8, this takes $\mathcal{O}(rs^2)$ time and $\mathcal{O}(rs)$ space.

- Compute the automaton $\mathcal{A}_{pre^*}$ accepting $pre^*(Rep\,\Gamma^*)$ using the information gathered in the previous step. This takes $\mathcal{O}(r)$ time and space. A configuration $\langle p, w \rangle$ violates φ exactly if $\mathcal{A}_{pre^*}$ accepts $\langle (p, q_0), w \rangle$. $\mathcal{A}_{pre^*} = ((P \times Q) \cup \{s\}, \Gamma, \rightarrow_{\mathcal{A}}, P \times Q, \{s\})$ has $\mathcal{O}(s)$ states and $\mathcal{O}(rs)$ transitions.

The complexity of this procedure, which solves problems (III) (whether the initial configuration satisfies φ) and (IV) (the global model-checking problem), is dominated by the time needed to compute the repeating heads and the space needed to store $\mathcal{A}_{pre^*}$. In the *post** method (see Section 3.2.3) we can also solve problem (III) like this:

- Let $\mathcal{A}$ be the automaton which accepts the configuration $\langle (p_0, q_0), w_0 \rangle$ of $\mathcal{BP}$. This automaton has s initial states and $|w_0|$ transitions and non-initial states.

- Let n_2 be the number of different heads $\langle p, \gamma \rangle$ for which there is a rule $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle \in \Delta$. According to Theorem 3.5, $\mathcal{A}_{post^*}$ can be constructed in $\mathcal{O}(rs \cdot (|w_0| + n_Q n_2))$ time and space.

- Using the information provided in the previous step we can construct the (partial) head reachability graph and search it in $\mathcal{O}(rs)$ time and space. The initial configuration violates φ exactly if a repeating head is found.

This complexity of this second procedure is dominated by the time and space for $\mathcal{A}_{post^*}$. We obtain the following results:

Theorem 3.9 *The model-checking problem for the initial configuration and the global model-checking problem can be solved in* $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |\mathcal{B}|^3)$ *time and* $\mathcal{O}(|P| \cdot |\Delta| \cdot |\mathcal{B}|^2)$ *space using the pre* method. The problem for the initial configuration can be solved in* $\mathcal{O}(|P| \cdot |\Delta| \cdot (|w_0| + n_2) \cdot |\mathcal{B}|^3)$ *time and space using the post* method.*

To solve problem (V), the global model-checking problem for reachable configurations, we need some additional steps:

- In $\mathcal{A}_{pre^*}$, rename the states (p, q_0) into p for all $p \in P$ and take P as the new set of initial configurations. Define n_2 as above.

- Compute *post** (with respect to $\mathcal{P}$, not $\mathcal{BP}$) for the initial configuration $\{\langle p_0, w_0 \rangle\}$ Then, according to Theorem 3.5, $\mathcal{A}_{post^*}$ can be computed in $\mathcal{O}(n_P n_\Delta(|w_0| + n_2))$ time and space. $\mathcal{A}_{post^*}$ has $m := n_P + |w_0| + n_2$ many states and $\mathcal{O}(n_P n_\Delta(|w_0| + n_2))$ many transitions.

- Compute the intersection of $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$. The resulting automaton $\mathcal{A}_{inter}$ accepts the set of reachable configurations violating φ. The transitions of $\mathcal{A}_{inter}$ are computed as follows:

  If (q, γ, q') in $\mathcal{A}_{pre^*}$ and (r, γ, r') in $\mathcal{A}_{post^*}$,
  then $((q, r), \gamma, (q', r'))$ in $\mathcal{A}_{inter}$.

  This intersection can be computed more efficiently if the following trick is employed: First all the transitions (r, γ, r') in $\mathcal{A}_{post^*}$ are sorted into buckets labelled γ. Then every transition of $\mathcal{A}_{pre^*}$ is multiplied with the transitions in the respective bucket. The number of transitions in each bucket is at most m^2. Hence, the intersection can be computed in $\mathcal{O}(rsm^2)$ time and space. Alternatively, we can create buckets for the transitions of $\mathcal{A}_{pre^*}$ which has s many states; we then obtain $\mathcal{O}(s^2 n_P n_\Delta(|w_0| + n_2))$ time and space for the intersection.

Theorem 3.10 *The global model-checking problem for reachable configurations can be solved in*

$$\begin{array}{rl} & \mathcal{O}(|P| \cdot |\Delta| \cdot |\mathcal{B}|^3 \cdot (|P| + |w_0| + n_2)^2) \textit{ time} \\ \textit{and} & \mathcal{O}(|P| \cdot |\Delta| \cdot |\mathcal{B}|^2 \cdot (|P| + |w_0| + n_2)^2) \textit{ space} \end{array}$$

or in

$$\begin{array}{rl} & \mathcal{O}(|P|^3 \cdot |\Delta| \cdot |\mathcal{B}|^3 \cdot (|w_0| + n_2)) \textit{ time} \\ \textit{and} & \mathcal{O}(|P|^3 \cdot |\Delta| \cdot |\mathcal{B}|^2 \cdot (|w_0| + n_2)) \textit{ space.} \end{array}$$

Conclusion The work in this section evolved from the model-checking algorithm in [9], which followed the same automata-theoretic approach using *pre** to solve problems (III) and (IV). Proposition 3.1 is taken from there; its proof has been repeated (and rephrased) here for the sake of completeness. The contribution of the algorithms presented here is twofold: the *pre** method is substantially better than the one from [9] which uses either $O(|\mathcal{BP}|^5)$ time and $O(|\mathcal{BP}|^2)$ space, or $O(|\mathcal{BP}|^3)$ time and space. The *post** method is new and often much faster in practice (see Section 5.5).

In [22] another algorithm was presented for deciding if the initial configuration satisfies a given LTL property, i.e. problem (III). (The problem of obtaining a representation for the solution of (IV) was not discussed.) The algorithm takes cubic time and also cubic space in the size of the pushdown system. More precisely, it is based on a saturation routine which requires $\Theta(|\mathcal{P}|^3 \cdot |\mathcal{B}|^3)$ space. Observe that the space consumption is Θ, and not O.

Since the algorithms presented here are quadratic in $|P|$ and only linear in $|\Delta|$, they specialize nicely to linear time and space for Basic Process Algebras [31] which are essentially pushdown systems with only one control location. More recently, LTL model-checking procedures for Hierarchical State Machines, a model equally expressive as pushdown systems, have been proposed [1, 6]. For a more detailed discussion of these, see Section 3.1.5.

3.3 Symbolic algorithms

Earlier on, pushdown systems were discussed as a model for sequential programs with variables of finite data types (see Section 2.3). We noticed that for this to be practical we require a succinct representation for the system because its size becomes exponential in the number of program variables. To this end, we introduced the concept of symbolic pushdown systems in which sets of rules are expressed as relations (for which we hope to find efficient representations). This approach is hybrid in the sense that we do not encode the whole system as one large relation, but partition it into multiple smaller relations. These come in the form of explicit rules to each of which a (symbolic) relation is attached, see the discussion in Section 2.3.

This section discusses versions of the model-checking algorithms which take advantage of such efficient representations. As a data structure for these representations we use Binary Decision Diagrams (see Section 2.4). In principle, however, the approach is amenable to any data structure which represents sets and provides efficient implementations for standard set operations like union, intersection, and existential quantification (see below).

For the rest of the section, we fix a symbolic pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ whose control locations and stack alphabet have the form

$$P = P_0 \times G \qquad \text{and} \qquad \Gamma = \Gamma_0 \times L\ .$$

Since we use BDDs to represent elements and subsets of G and L, we can assume that G and L are the cross product of m_G and m_L boolean variables, respectively:

$$G = \{0,1\}^{m_G} \qquad \text{and} \qquad L = \{0,1\}^{m_L}$$

Symbolic automata The reachability algorithms from Section 3.1 operate on regular sets of configurations encoded by finite automata (Definition 3.2). To adapt these algorithms to the symbolic case, we need an appropriate data structure for the automata. We refer to the data structure presented below as *symbolic automata*.

Let $\mathcal{A} = (Q_0 \times G, \Gamma, \rightarrow, P_0 \times G, F)$, where $Q_0 \supseteq P_0$, be a $\mathcal{P}$-automaton. If $q, q' \in Q_0$ and $\gamma \in \Gamma_0$, then we let

$$q \xrightarrow[{[R]}]{\gamma} q'$$

(where $R \subseteq G \times L \times G$) denote the set of transitions

$$\{\, (q, g) \xrightarrow{(\gamma,l)} (q, g') \mid (g, l, g') \in G \times L \times G \,\}.$$

If we hope that BDDs (or some other encoding) compactly represent elements of G and L in $\mathcal{P}$, it seems justified to hope that the same type of encoding is succinct for the relation R above. Therefore, we choose to represent $\rightarrow$ as a list of so-called *symbolic transitions* $q \xrightarrow[{[R]}]{\gamma} q'$ as above, i.e. triples from $Q_0 \times \Gamma_0 \times Q_0$ to which a BDD is attached.

3.3.1 Operations on sets and BDDs

The adaptation of the model-checking algorithms from Sections 3.1 and 3.2 is achieved by set-theoretic operations on the relations attached to symbolic rewriting rules of $\mathcal{P}_S$ and the transitions of $\mathcal{P}_S$-automata. Since the relations will be represented by BDDs, it is worth having a look at which these operations are and what their BDD counterparts are.

A subset of G can be represented by a BDD with m_G variables. Likewise, a BDD with m_L variables can represent a subset of L. To represent multi-dimensional relations whose components can be elements of either G or L, we need BDDs where the individual components are represented by pairwise disjoint sets of BDD variables. We shall call these sets the *signature* of the BDD. More precisely, suppose that a k-dimensional relation R is expressed by a BDD $\mathcal{R}$ in such a way that the i-th component ($1 \leq i \leq k$) of the tuples in R is represented by a set BDD variables V_i. Then we define $\mathit{sig}(\mathcal{R}) := (V_1, \ldots, V_k)$.

Union and Intersection Standard set operations like union and intersection have immediate counterparts in BDD operations; e.g. $R_1 \cup R_2$ and $R_1 \cap R_2$ correspond to $\mathcal{R}_1 \vee \mathcal{R}_2$ and $\mathcal{R}_1 \wedge \mathcal{R}_2$ if $\mathcal{R}_1$ and $\mathcal{R}_2$ are BDDs for R_1 and R_2, respectively. In order for the result to be unambiguous, the signatures of both BDDs need to be the same.

Join Another important operation is the *join* between two relations. For instance, if $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$, we may want to compute the set

$$R_3 = \{\, (x, z) \mid \exists y \in S_2 \colon (x, y) \in R_1 \wedge (y, z) \in R_2 \,\}.$$

The corresponding BDD operation involves conjunction and existential abstraction. If v is a BDD variable and $\mathcal{R}$ a BDD, then $\exists v \colon \mathcal{R}$ denotes $(\mathcal{R} \wedge \neg v) \vee (\mathcal{R} \wedge v)$. We extend this notation so that $\exists V \colon \mathcal{R}$ means existential abstraction over all variables in the set V. Assume that $\mathcal{R}_1$ and $\mathcal{R}_2$ are BDDs for R_1 and R_2 with $\mathit{sig}(\mathcal{R}_1) = (V_1, V_2)$ and $\mathit{sig}(\mathcal{R}_2) = (V_2, V_3)$. A

BDD $\mathcal{R}_3$ for R_3 is then obtained by computing

$$\mathcal{R}_3 = \exists V_2\colon (\mathcal{R}_1 \wedge \mathcal{R}_2).$$

The signature of $\mathcal{R}_3$ is (V_1, V_3).

Relabelling In the example for joins, imagine that R_3 holds the same type of information as R_1. In that case it would be practical if $\mathcal{R}_3$ had the same signature as $\mathcal{R}_1$. We use an operation called *relabelling* for this:

Generally speaking, if $W = \{w_1, \ldots, w_j\}$ and $Z = \{z_1, \ldots, z_j\}$ are sets of BDD variables and $\mathcal{R}$ is a BDD with $\mathit{sig}(\mathcal{R}) = (V_1, \ldots, V_{l-1}, W, V_{l+1}, \ldots, V_k)$, then we let $\mathcal{R}[W \to Z]$ denote the BDD which is obtained from $\mathcal{R}$ by relabelling all nodes labelled by $w_i,\ 1 \leq i \leq j$, to z_i. The signature of $\mathcal{R}[W \to Z]$ is $(V_1, \ldots, V_{l-1}, Z, V_{l+1}, \ldots, V_k)$. In the example, the operation $\mathcal{R}_3[V_3 \to V_2]$ would yield a BDD with signature (V_1, V_2).

If multiple components of $\mathcal{R}$ are changed simultaneously, we write a comma-separated list, e.g. $\mathcal{R}[W_1 \to Z_1,\ W_2 \to Z_2, \ldots]$.

In the following, let G_0, G_1, G_2 be disjoint sets of m_G BDD variables each, and let L_0, L_1, L_2, L_3 be disjoint sets of m_L BDD variables each. If we have $\langle p, \gamma\rangle \underset{[R]}{\hookrightarrow} \langle p', \gamma_1, \ldots, \gamma_n\rangle$, where $0 \leq n \leq 2$, we assume that a BDD which represents R has the signature $(G_0, L_0, G_1, L_1, \ldots, L_n)$. For both pre^* and post^*, the abstract algorithms will be presented in terms of set operations whereas the concrete algorithms will show their BDD counterparts. For the latter, we will in fact remove the distinction between relations and the BDDs representing them because each relation is represented by a BDD with one particular signature which is therefore unambiguous.

3.3.2 Computing *pre**

Let $\mathcal{A} = (Q_0 \times G, \Gamma, \to_0, P_0 \times G, F)$ be a $\mathcal{P}$-automaton for the regular set of configurations C. Without loss of generality, we assume that $\mathcal{A}$ has no transitions ending in an initial state. Section 3.1.1 showed a saturation procedure which computes $\to\ \supseteq\ \to_0$, the transition relation of the automaton recognizing $\mathit{pre}^*(C)$:

If $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ and $p' \xrightarrow{w}{}^* q$, then add $p \xrightarrow{\gamma} q$.

For symbolic representations, we instantiate the rule as follows:

(i) If $\langle p, \gamma \rangle \xhookrightarrow[\;[R]\;]{} \langle p', \varepsilon \rangle$, then add $p \xrightarrow[\;[R]\;]{\gamma} p'$.

(ii) If $\langle p, \gamma \rangle \xhookrightarrow[\;[R]\;]{} \langle p', \gamma' \rangle$ and $p' \xrightarrow[\;[R_1]\;]{\gamma'} q$, then add $p \xrightarrow[\;[R']\;]{\gamma} q$ where

$$R' = \{\, (g, l, g_1) \mid \exists g_0, l_1 \colon (g, l, g_0, l_1) \in R \wedge (g_0, l_1, g_1) \in R_1 \,\}.$$

(iii) If $\langle p, \gamma \rangle \xhookrightarrow[\;[R]\;]{} \langle p', \gamma'\gamma'' \rangle$ and $p' \xrightarrow[\;[R_1]\;]{\gamma'} q' \xrightarrow[\;[R_2]\;]{\gamma''} q$, then add $p \xrightarrow[\;[R']\;]{\gamma} q$ where

$$R' = \{\, (g, l, g_2) \mid \exists g_0, l_1, g_1, l_2 \colon (g, l, g_0, l_1) \in R \\ \wedge\ (g_0, l_1, g_1) \in R_1 \wedge (g_1, l_2, g_2) \in R_2 \,\}.$$

Notice that the operations required for cases (ii) and (iii) are joins between the relations R, R_1, and R_2.

Algorithm 5 in Figure 3.10 is an implementation of this instantiation of the saturation rule along the lines of Algorithm 1. We maintain two structures for sets of transitions, *rel* and *trans*. For each triple (q, γ, q'), where $q, q' \in Q_0$ and $\gamma \in \Gamma_0$, $\mathit{rel}(q, \gamma, q')$ and $\mathit{trans}(q, \gamma, q')$ represent a set of transitions which belong to $\rightarrow$. *trans* contains transitions that still need to be processed, and *rel* contains those that have already been processed.

Each relation in *rel* and *trans* is represented by a BDD with the signature (G_1, L_1, G_2). The relabelling operations ensure that all newly computed transitions have this common signature.

3.3.3 Computing post^*

Again, let $\mathcal{A} = (Q_0 \times G, \Gamma, \rightarrow_0, P_0 \times G, F)$ be a $\mathcal{P}$-automaton for a regular set of configurations C, and assume that $\mathcal{A}$ has no transitions leading to initial states. We modify $\mathcal{A}$ into an automaton with ε-moves which accepts $\mathit{post}^*(C)$. Section 3.1.2 showed a method which adds both states and transitions to $\mathcal{A}$, which we adapt to the symbolic case. Let $Q' \supseteq Q_0 \times G$ and $\rightarrow\ \supseteq\ \rightarrow_0$ contain the states and transitions of $\mathcal{A}_{\mathit{post}^*}$, the $\mathcal{P}$-automaton for $\mathit{post}^*(C)$. We proceed as follows:

- For each symbolic rule $\langle p, \gamma \rangle \xhookrightarrow[\;[R]\;]{} \langle p', \gamma'\gamma'' \rangle$ we add to Q' a set of new states $\{q_{p',\gamma'}\} \times (G \times L)$.
- The transition relation $\rightarrow$ becomes a little complicated since we must adjust it to the new states and to ε-transitions. For instance, we write $p \xrightarrow[\;[R]\;]{\varepsilon} q_{p',\gamma'}$, where $R \subseteq G \times (G \times L)$ to denote the set of transitions

$$\{\, (p, g) \xrightarrow{\varepsilon} (q_{p',\gamma'}, g', l') \mid (g, g', l') \in R \,\}.$$

Algorithm 5

Input: a symbolic pushdown system $\mathcal{P} = (P_0 \times G, \Gamma_0 \times L, \Delta, c_0)$;
a symbolic $\mathcal{P}$-Automaton $\mathcal{A} = (Q_0 \times G, \Gamma_0 \times L, \rightarrow_0, P_0 \times G, F)$
without transitions into initial states

Output: the transition relation $\rightarrow$ of $\mathcal{A}_{pre^*}$

procedure $add\_trans\ (q, \gamma, q', R)$
begin
  $trans(q, \gamma, q') := (trans(q, \gamma, q') \cup R) \setminus rel(q, \gamma, q')$
end

$rel := \emptyset;\ trans := \rightarrow_0;\ \Delta' := \emptyset;$
for all $\langle p, \gamma \rangle \xhookrightarrow[{[R]}]{} \langle p', \varepsilon \rangle \subseteq \Delta$ **do**
  $add\_trans(p, \gamma, p', R[G_0 \rightarrow G_1, L_0 \rightarrow L_1, G_1 \rightarrow G_2]);$
while $trans \neq \emptyset$ **do**
  pick (q, γ, q') s.t. $trans(q, \gamma, q') \neq \emptyset;$
  $R := trans(q, \gamma, q');$
  $trans(q, \gamma, q') := \emptyset;$
  $rel(p', \gamma', q) := rel(p', \gamma', q) \cup R;$
  for all $\langle p', \gamma' \rangle \xhookrightarrow[{[R']}]{} \langle q, \gamma \rangle \subseteq (\Delta \cup \Delta')$ **do**
    $add\_trans(p', \gamma', q', (\exists G_1 \cup L_1 \colon R \wedge R')\,[G_0 \rightarrow G_1, L_0 \rightarrow L_1]);$
  for all $\langle p', \gamma' \rangle \xhookrightarrow[{[R']}]{} \langle q, \gamma\gamma'' \rangle \subseteq \Delta$ **do**
    $R_{tmp} := (\exists G_1 \cup L_1 \colon R \wedge R')\,[G_2 \rightarrow G_1, L_2 \rightarrow L_1];$
    $\Delta' := \Delta' \cup \langle p', \gamma' \rangle \xhookrightarrow[{[R_{tmp}]}]{} \langle q', \gamma'' \rangle;$
    for all $q'' \in P_0$ s.t. $rel(q', \gamma'', q'') \neq \emptyset$ **do**
      $R'' := rel(q', \gamma'', q'');$
      $add\_trans(p', \gamma', q'', (\exists G_1 \cup L_1 \colon R_{tmp} \wedge R'')\,[G_0 \rightarrow G_1, L_0 \rightarrow L_1]);$
return rel

Figure 3.10: pre^* for the symbolic case.

The saturation rule from Section 3.1.2 can roughly be summarized as follows:

If $p \xrightarrow{\gamma}{}^{*} q$ and $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$,
add transitions to create a path $p' \xrightarrow{w}{}^{*} q$.

Earlier we noted that $p \xrightarrow{\gamma}{}^{*} q \iff p \xrightarrow{\gamma} q \vee \exists q' \colon p \xrightarrow{\varepsilon} q' \wedge q' \xrightarrow{\gamma} q$. Analogously, we write $p \xrightarrow[{[R]}]{\gamma}{}^{*} q$ below if for every triple $(g, l, g') \in R$ we have

$$\begin{aligned}(p, g) \xrightarrow{(\gamma,l)} (q, g') \vee \exists q'', g'' \colon (p, g) \xrightarrow{\varepsilon} (q'', g'') \xrightarrow{(\gamma,l)} (q, g')\\ \vee\ \exists p', \gamma', g'', l'' \colon (p, g) \xrightarrow{\varepsilon} (q_{p',\gamma'}, g'', l'') \xrightarrow{(\gamma,l)} (q, g')\end{aligned}$$

We instantiate the saturation rule for the following three cases, in which we assume that $q \in Q_0$:

(i) If $\langle p, \gamma\rangle \underset{[R]}{\hookrightarrow} \langle p', \varepsilon\rangle$ and $p \xrightarrow[{[R']}]{\gamma}{}^{*} q$, then add $p' \xrightarrow[{[R'']}]{\varepsilon} q$, where

$$R'' = \{\, (g', g'') \mid \exists (g, l, g') \in R,\ (g, l, g'') \in R' \,\}.$$

(ii) If $\langle p, \gamma\rangle \underset{[R]}{\hookrightarrow} \langle p', \gamma'\rangle$ and $p \xrightarrow[{[R']}]{\gamma}{}^{*} q$, then add $p' \xrightarrow[{[R'']}]{\gamma'} q$, where

$$R'' = \{\, (g', l', g'') \mid \exists (g, l, g', l') \in R,\ (g, l, g'') \in R' \,\}.$$

(iii) If $\langle p, \gamma\rangle \underset{[R]}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle$ and $p \xrightarrow[{[R']}]{\gamma}{}^{*} q$, then add $p' \xrightarrow[{[R'']}]{\gamma'} q_{p',\gamma'} \xrightarrow[{[R''']}]{\gamma''} q$, where

$$\begin{aligned}R'' &= \{\, (g', l', (g', l')) \mid \exists (g, l, g', l', l'') \in R,\ (g, l, g'') \in R' \,\}\\ R''' &= \{\, ((g', l'), l'', g'') \mid \exists (g, l, g', l', l'') \in R,\ (g, l, g'') \in R' \,\}.\end{aligned}$$

Three additional but very similar cases arise when q has the form $q_{p',\gamma'}$, so that the set R' in the three cases above is a subset of $G \times L \times (G \times L)$.

Again, notice that R'' and R''' in cases (i) to (iii) are the result of join operations. Algorithm 6 in Figure 3.11 is a *post** procedure similar to Algorithm 2. Again we use *trans* and *rel* to store transitions that we need to examine. The procedure *add_trans* is the same as in Algorithm 5.

We have three different types of states in Q': initial states, states added by the *post** procedure, and other non-initial states. For efficiency reasons, the signatures of the BDDs used to represent the transitions vary with the types of states they connect. For a triple (q, γ, q'), where $\gamma \in \Gamma_0$, the signatures of $rel(q, \gamma, q')$ and $trans(q, \gamma, q')$ are:

$$\begin{array}{ll} (G_0, L_0, G_2) & \text{if } q \in P_0,\ q' \in (Q_0 \setminus P_0); \\ (G_0, L_0, G_2, L_3) & \text{if } q \in P_0 \text{ and } q' = q_{p',y'} \text{ is a new state;} \\ (G_1, L_1, L_0, G_2, L_3) & \text{if } q, q' \text{ are new states;} \\ (G_1, L_1, L_0, G_2) & \text{if } q \text{ is a new state and } q' \in (Q_0 \setminus P_0); \\ (G_1, L_0, G_2) & \text{if } q, q' \in (Q_0 \setminus P_0). \end{array}$$

The signatures of $rel(p, \varepsilon, q)$ and $trans(p, \varepsilon, q)$ are:

$$\begin{array}{ll} (G_0, G_1) & \text{if } q \in (Q_0 \setminus P_0); \\ (G_0, G_1, L_1) & \text{if } q = q_{p',\gamma'} \text{ is a new state;} \end{array}$$

The relabelling operations in Algorithm 6 ensure that all newly computed transitions have the appropriate signatures.

In line 20, the terms $G_0 \equiv G_2$ and $L_0 \equiv L_3$ denote BDDs which express pairwise equivalence between the variables in G_0 and G_2, and L_0 and L_3, respectively.

3.3.4 Problems of the symbolic approach

Section 3.2 presented a method for computing the repeating heads. The method involves constructing a directed graph $\mathcal{G}$ whose nodes are the heads of the pushdown system, and whose edges are labelled by 0 or 1. A compact representation of the graph in the symbolic case is straightforward; if we let $(p, \gamma) \xrightarrow[{[R]}]{b} (p', \gamma')$ stand for the set of edges

$$\{\, ((p, g), (\gamma, l)) \xrightarrow{b} ((p', g'), (\gamma', l')) \mid (g, l, g', l') \in R \,\},$$

then the edges of $\mathcal{G}$ can be stored as a list of such sets.

While the mere construction of $\mathcal{G}$ can be carried out efficiently, the decomposition into strongly connected components is more difficult. The algorithm used to find the SCCs performs a depth-first search of the graph. This involves looking at each individual node in the graph and following each outgoing edge recursively. Looking the nodes and edges individually, however, destroys all advantages of symbolic representations, whose strength lies in processing large groups of nodes with a single operation. This is actually a well-known problem also encountered when model-checking finite-state

Algorithm 6

Input: a symbolic pushdown system $\mathcal{P} = (P_0 \times G, \Gamma_0 \times L, \Delta, c_0)$;
a symbolic $\mathcal{P}$-Automaton $\mathcal{A} = (Q_0 \times G, \Gamma_0 \times L, \rightarrow_0, P_0 \times G, F)$
recognizing C without transitions into initial states or ε-moves

Output: a $\mathcal{P}$-automaton recognizing $post^*(C)$

$Q' := Q_0 \times G$;
for all $q \in Q_0,\ \gamma \in \Gamma_0,\ q' \in Q_0$ **do**
  if $q \xrightarrow[{[R]}]{\gamma}_0 q'$ **then**
    if $q \in P_0$ **then** $trans(q, \gamma, q') := R$; **else** $rel(q, \gamma, q') := R$;
for all $\langle p, \gamma\rangle \xhookrightarrow[{[R]}]{} \langle p', \gamma'\gamma''\rangle \in \Delta$ **do**
  $Q' := Q' \cup \{q_{p',\gamma'}\} \times (G \times L)$;
while $trans \neq \emptyset$ **do**
  pick (p, γ, q) s.t. $trans(p, \gamma, q) \neq \emptyset$;
  $R := trans(p, \gamma, q)$;
  $trans(p, \gamma, q) := \emptyset$;
  $rel(p, \gamma, q) := rel(p, \gamma, q) \cup R$;
  if $\gamma \neq \varepsilon$ **then**
    for all $\langle p, \gamma\rangle \xhookrightarrow[{[R']}]{} \langle p', \varepsilon\rangle \subseteq \Delta$ **do**
      $R'' := (\exists G_0, L_0\colon R \wedge R')\,[G_1 \to G_0, G_2 \to G_1, L_3 \to L_1]$;
      $add\_trans(p', \varepsilon, q, R'')$;
    for all $\langle p, \gamma\rangle \xhookrightarrow[{[R']}]{} \langle p', \gamma'\rangle \subseteq \Delta$ **do**
      $R'' := (\exists G_0, L_0\colon R \wedge R')\,[G_1 \to G_0, L_1 \to L_0]$;
      $add\_trans(p', \gamma', q, R'')$;
    for all $\langle p, \gamma\rangle \xhookrightarrow[{[R']}]{} \langle p', \gamma'\gamma''\rangle \subseteq \Delta$ **do**
      $R'' := (\exists G_0, L_0, L_2, G_2, L_3\colon R \wedge R')[G_1 \to G_0, L_1 \to L_0]$
        $\wedge\ (G_0 \equiv G_2) \wedge (L_0 \equiv L_3)$;
      $add\_trans(p', \gamma', q_{p',\gamma'}, R'')$;
      $R_{tmp} := (\exists G_0, L_0\colon R \wedge R')\,[L_2 \to L_0]$;
      $rel(q_{p',\gamma'}, \gamma'', q) := rel(q_{p',\gamma'}, \gamma'', q) \cup R_{tmp}$;
      for all p'' s.t. $rel(p'', \varepsilon, q_{p',\gamma'}) \neq \emptyset$ **do**
        $add\_trans(p'', \gamma'', q, \exists G_1, L_1\colon R_{tmp} \wedge rel(p'', \varepsilon, q_{p',\gamma'}))$;
  else
      if $R_{tmp} \not\subseteq eps(p', q)$ **then**
        for all γ'', q' s.t. $rel(q', \gamma'', q') \neq \emptyset$ **do**
          $add\_trans(p', \gamma'', q', \exists G_1, L_1\colon R \wedge rel(q', \gamma'', q'))$;
return $(Q', \Gamma_0 \times L, rel, P_0 \times G, F)$

Figure 3.11: $post^*$ for the symbolic case

systems. Section 5.2 discusses some possible approaches to alleviate the problem. None of the known methods can avoid a higher than linear complexity in the size of the graph, which means that the model-checking problem for symbolic systems has a worse asymptotic complexity than for normal systems. In practice however, the advantages of having a more succinct representation more than make up for this disadvantage.

In Section 3.1.6 we discussed a method for reconstructing a witness path from a configuration $c \in C$ to $c' \in post^*(C)$ by evaluating information gathered during the computation of $post^*(C)$. This method involves labelling each transition with the rewrite rule that led to its addition. While this approach can be transferred to the symbolic case, it is not necessarily desirable to do so. Earlier on, we saw that the signatures of the BDDs in $\rightarrow$ can have up to five dimensions. If we add the information for the reconstruction, we get another two to four dimensions, which can not only require a large amount of memory but also time, because subsequent operations in $post^*$ have to operate on larger BDDs. A possible solution is to annotate the transitions with less information which still allows to reconstruct a path, but at the expense of some more computation during the reconstruction. This is discussed in Section 5.3.

Chapter 4

Experiments with Moped

The various model-checking algorithms of Chapter 3 have been implemented in a model-checking tool called Moped.[1] This chapter first presents some simple examples to demonstrate how the tool can be used, and then reports on experimental results with a number of fairly large examples.

The input to the Moped tool consists of a system and a property. The system is either directly specified as a pushdown system (in symbolic or explicit form) or indirectly as a Boolean Program. The tool supports binary decision diagrams (BDDs) as a representation for symbolic pushdown systems; operations on BDDs are implemented using the CUDD package [38]. The property is either a reachability property or an LTL formula. The precise syntax of the input languages and the properties is described in the Appendix. Moreover, the user can influence the behaviour of the checker with a number of options, e.g. by determining whether the pre^* or the $post^*$ method should be used, and a number of other things. The options are listed in Section A.4 of the Appendix, and some of them are discussed in greater detail in Chapter 5.

The output of the checker is a yes/no answer – the given property either holds for the system or it does not hold. Moreover, the answer may be supported by a witness path (if a reachability property holds) or a counterexample (if an LTL property does not hold). When a Boolean Program is checked for reachability, the witness path may be printed in a number of different formats in order to interact with other tools.

[1] The early stages of the work on Moped, which consisted of implementing the algorithms developed in [9], were carried out by David Hansel.

4.1 Examples

Moped supports two separate languages for specifying the system – (symbolic) pushdown systems and Boolean Programs (which are internally translated into pushdown systems). In this section, we show how to model the 'plotter' and 'lock/unlock' examples from Chapter 2 in these languages, and how to use Moped to check some properties of them. In these examples, input made by the user on the command line is marked by prefixing it with a $ sign.

4.1.1 The plotter example

The 'plotter' example was introduced in Section 2.3 (see Figure 2.1). In the following, we model this example in the language for pushdown systems.

Pushdown systems are defined as quadruples $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ where P is the set of control locations, Γ the stack alphabet, Δ the set of rewrite rules, and c_0 the initial configuration. In Moped, a pushdown system is specified by just the initial configuration c_0 and a list of rewrite rules which make up Δ. The control locations and the stack alphabet are defined implicitly by the presence of identifiers in c_0 and Δ, i.e. whenever an identifier appears in a place where Moped expects a control location or a stack alphabet, the checker's internal representation of $\mathcal{P}$ contains a control location or stack symbol with the respective name. Each rewrite rule may have a 'label', which simply serves as a comment and is printed in witness/counterexample paths which fire the rule. Like in the algorithms from Chapter 3, the right-hand sides of the rules are restricted to two symbols.

The example in Figure 4.1 shows the translation of the plotter example from Figure 2.3 into the input language of Moped. The first line fixes the initial configuration as $\langle q, main_0 \rangle$, all other lines define the rewrite rules. Everything following a `#` sign until the end of the line is treated as a comment.

We proposed that a desirable property of the plotter is that an upwards movement is never immediately followed by a downward movement movement. This can be expressed as an LTL property (see Section 2.5). In our model, this can be verified by

```
$ moped plot.pds '[](up0 -> (!down0 U up0 || right0))'
```

Here, `[]` is Moped's rendering for the $\Box$ (always) operator, `->` means

```
(q <main0>)

# procedure m
q <m0> --> q <m3>              q <m0> --> q <m7>
q <m3> --> q <s0 m4>           q <m4> --> q <right0 m5>
q <m5> --> q <m1>              q <m5> --> q <m6>
q <m6> --> q <m0 m1>           q <m7> --> q <up0 m8>
q <m8> --> q <m0 m2>           q <m2> --> q <down0 m1>
q <m1> --> q <>

# procedure s
q <s0> --> q <s2>              q <s0> --> q <s3>
q <s2> --> q <up0 s4>          q <s3> --> q <>
q <s4> --> q <m0 s5>           q <s5> --> q <down0 s1>
q <s1> --> q <>

# procedure main
q <main0> --> q <s0 main1>     q <main1> --> q <>

# procedures up, down, right
q <up0> --> q <>               q <down0> --> q <>
q <right0> --> q <>
```

Figure 4.1: The plotter example in Moped's syntax.

implication, and `||` stands for disjunction [2] The atomic propositions allowed for pushdown systems are the control locations and stack symbols which occur in the pushdown system. For instance, in the formula above the atomic property `up0` is true for configurations where the stack symbol `up0` is on the top of the stack.

With the invocation above, Moped would decide that the property holds and would reply

```
YES
```

Alternatively, we might want to check whether the program always terminates, which we can express by asking whether `main1` is reached in *every* execution. Moped would find that this is not the case because the procedures `m` and `s` can recursively call each other forever:

```
$ moped plot.pds '<>main1'
NO.
--- START ---
q <main0>
q <s0 main1>
q <s2 main1>
q <up0 s4 main1>
q <s4 main1>
q <m0 s5 main1>
--- LOOP ---
q <m3 s5 main1>
q <s0 m4 s5 main1>
q <s2 m4 s5 main1>
q <up0 s4 m4 s5 main1>
q <s4 m4 s5 main1>
q <m0 s5 s4 m4 s5 main1>
```

The counterexample is a run which begins with the configuration printed after `START` and then repeats the part after `LOOP` forever. Notice that the configuration at the end of the loop has the same head as the one immediately before the loop begins. Indeed, `main1` is never reached in this run.

[2] Actually, Moped uses Spin [25] to translate LTL formulas into Büchi automata; thus, the format of LTL formulas is the same as Spin's, and it is shown in detail in the Appendix.

4.1.2 The lock/unlock example

In Section 2.3 we also considered a program which uses boolean variables to determine whether a lock is currently held by the program (see Figure 2.4). We argued that such variables should be represented in some compact fashion, for instance using BDDs. As mentioned before, Moped offers two input languages for such systems. In this section, we show how to use the input language for symbolic pushdown systems to model the lock/unlock example; in Section 4.1.3 we show how to do the same with the language for Boolean Programs.

In order to specify a symbolic pushdown system, the user must first define a domain of global and local variables (called G and L, respectively). We then take $P := P_0 \times G$ and $\Gamma := \Gamma_0 \times L$ where P_0 and Γ_0 are the 'explicit' control locations and stack symbols which are (again) implicitly defined by their presence in the input. Variables can be of types boolean and integer (the latter restricted to finite ranges), or arrays of both types. When variables are defined, each rule listed in the input file becomes a symbolic rule. The relations associated with the symbolic rules are given by boolean expressions over the global and local variables, i.e. the relations are understood to be the set of valuations which make the expressions true. This is exemplified by the system in Figure 4.2 which is essentially Moped's rendition of the pushdown system in Figure 2.6.

The first three lines define the variables. The line

```
global bool l,r;
```

establishes the two global boolean variables, `l` and `r`. Moreover, the user can assign a domain of local variables to each explicit stack symbol. The effect of this is that if a set of variables V is assigned to symbol γ, then symbolic rules involving γ may talk about the variables in V. In the example, the domains are fixed in the next two lines:

```
local (g0) bool x;
local (main0,main1,...,main5) bool a,b;
```

This assigns the variable `x` to those (explicit) stack symbols which correspond to function `g` and the variables `a` and `b` to those of function `main`. Notice that the asymptotic complexity of the associated verification problems depends on the size of the stack alphabet which is the cross product between the explicit stack symbols and the domains of local variables assigned to them. Therefore,

```
global bool l,r;
local (g0) bool x;
local (main0,main1,main2,main3,main3a,main4,main5) bool a,b;

(q <main0>)

q <lock0> --> q <err>          (l & l')
q <lock0> --> q <lock1>        (!l & !l')
q <lock1> --> q <lock2>        (l')
q <lock2> --> q <>             (l'==l)
q <err> --> q <err>            (l'==l)

q <unlock0> --> q <err>        (!l & !l')
q <unlock0> --> q <unlock1>    (l & l')
q <unlock1> --> q <unlock2>    (l'==l)
q <unlock2> --> q <>           (!l')

q <g0> --> q <>                ((l'==l) & r'^x)

q <main0> --> q <main1>        (!l' & !a' & (b'==b))
q <main1> --> q <main2>        ((l'==l) & (a'==a) & (b'==b))
q <main2> --> q <lock0 main3>  ((l'==l) & (a''==a) & (b''==b))
q <main3> --> q <g0 main3a>    ((l'==l) & (x'==a)
                                        & (a''==a) & (b''==b))
q <main3a> --> q <main4>       ((l'==l) & (a'==a) & (b'==r))
q <main4> --> q <unlock0 main5>((l'==l) & (a''==a) & (b''==b))
q <main5> --> q <>             (l'==l)
```

Figure 4.2: The lock/unlock example in Moped's syntax.

only the size of the largest domain of local variables matters. The ability to specify multiple domains of local variables is thus a syntactic convenience, but does not increase the asymptotic complexity of the associated verification problems.

After the variables, the initial configuration is fixed as $\langle q, main_0 \rangle$. Since no expression is given, the initial variable values are undetermined, i.e. effectively we have multiple initial configurations. The initial configuration is followed by the symbolic rules, each of which is accompanied by a boolean expression. Let us just consider one of them since all are very similar:

```
q <main3> --> q <g0 main3a>
      ((l'==l) & (x'==a) & (a''==a) & (b''==b))
```

In terms of the example program in Figure 2.4, this symbolic rule models a call from `main` to function `g`, passing the value of `main`'s local variable `a` as an argument to `g`. The parameter is called `x` within `g`. In the expression above, unprimed identifiers (like `l, a, b`) refer to the local and global variables on the left-hand side; observe that `a` and `b` are local variables associated with `main3`. Singly primed global variables (`l`′) are associated with the right-hand side control location. Singly primed local variables (`x`′) refer to the first right-hand side stack symbol (`g0` in this case), and doubly primed local variables (`a'', b''`) refer to the second stack symbol. Effectively, the rule stipulates that `main3` may be replaced with `g0` and `main3a` on the stack, provided that the global variable `l` retains its value, the variable `x` (of `g`) gets the value of `a` (of `main`), and the previous values of `a` and `b` are saved on the stack. There is no condition about the value of `r`, therefore any combination of pre- and post-values of `r` is allowed.

Suppose that we store the input in a file called `lock.pds`. Now, we might be interested in checking whether the program can cause an error, i.e. whether a lock occurs twice without an unlock in between, or an unlock occurs when no lock is set, in which case the example should push the symbol `err` to the top of the stack. The corresponding invocation of *Moped* would look like this: like this:

```
$ moped -r lock.pds q:err
```

Here, the `-r` argument signifies that we want Moped to perform a reachability analysis. Reachability formulas in Moped have the form *ctrlid*:*stackid* where *ctrlid* is the name of a control location and *stackid* is the name of a stack symbol. Since there is just one (explicit) control location `q`, we just have to check for `q:err`. The tool would recognize that no error can occur in the example and therefore respond with

```
NO.
```

Now, suppose one of the rules in `main` is replaced with another call to `lock`:

```
q <main1> --> q <lock0 main2> ((l'==l) & (a''==a) & (b''==b))
```

This would cause two lock operations in a row. If we store the changed file as `lock-error.pds`, another invocation of Moped detects the error:

```
$ moped -r lock-error.pds q:err
YES.
```

However, we might want to know how the error is actually reached. Adding `-t` to the options causes a witness path to be printed:

```
$ moped -rt lock-error.pds q:err
YES.
--- START ---
q (!l & !r) <main0 (!a & !b)>
q (!l & !r) <main1 (!a & !b)>
q (!l & !r) <lock0 main2 (!a & !b)>
q (!l & !r) <lock1 main2 (!a & !b)>
q (l & !r) <lock2 main2 (!a & !b)>
q (l & !r) <main2 (!a & !b)>
q (l & !r) <lock0 main3>
q (l & !r) <err main3>
[ target reached ]
```

The witness path shows an execution which enters the procedure `lock` twice; on the first occasion, `l` is set to true, on the second occasion an error is raised. Variable contents are printed as expressions; e.g. `(l & !r)` means that `l` is true and `r` is false. Global variables are printed after the control location, local variables along with the stack symbols (notice that `lock` has no local variables in the example). Moreover, to avoid cluttering up the output, Moped deliberately chooses not to output local variables of stack symbols to which the trace will never return (see `main3` in the example).

There is no explicit support in Moped to check for reachability of different categories of configurations. While in principle it would be easy to add support for different sorts of reachability queries, the ability to check for reachability of a 'head' $\langle p, \gamma \rangle$ is the most relevant for many practical problems. But even in the current state of the tool, one could for instance emulate regular languages as reachability targets by non-deterministically allowing the pushdown system to enter the simulation of a finite automaton which recognizes the languages by destructively reading the stack.

4.1.3 Lock/unlock example as a Boolean Program

For use in their SLAM project, Ball and Rajamani have created the language of Boolean Programs [3, 5]. Moped supports Boolean Programs as a second input language, which has two benefits: While Moped's original input language (see Sections 4.1.1 and 4.1.2) was designed to be as simple as possible while enabling the user to specify every possible symbolic pushdown system, its simplicity sometimes offers little comfort. Boolean Programs offer the user the more familiar-looking syntax of a program with assignments, procedures which can take arguments and return values etc, and hence allow a more natural specification for models derived from such systems. However, the Boolean Program syntax does not allow arrays or integer variables.

A flavour of the syntax is given by Figure 4.3 in which the lock/unlock example is formulated as a Boolean Program (see Figure 2.4 for the original program and Figure 4.2 for its formulation in Moped's other input language). A complete description of the syntax of the language is given in the Appendix.

Statements in Boolean Programs may be labelled; for instance, in Figure 4.3, the statement in procedure `error` is labelled `E`. In reachability queries, the user is expected to pass the name of a label. Mopedthen checks if there is any execution that leads to the statement with the given label. The label may be given in the form *function*: *label* or, omitting the function name, *label*. The first form resolves ambiguities if the given label occurs in multiple functions; if the second form is used, Moped automatically finds a function in which the label is defined.

In the example above, we may want to check whether an error may be raised by either `lock` or `unlock`. This is the case if and only if the label `E` in `error` is reachable. The corresponding invocation of Moped is as follows:

`$ moped -br lock.bp error:E` ⟨*other options*⟩

```
decl l;

void error ()
begin
  E: goto E;
end

void lock ()
begin
  if (l) then error(); fi
  l := T;
end

void unlock ()
begin
  if (!l) then error(); fi
  l := F;
end

bool g (x)
begin
  return !x;
end

void main ()
begin
  decl a,b;
  l,a := F,F;
  lock();
  b := g(a);
  unlock();
end
```

Figure 4.3: The lock/unlock example as a Boolean Program.

In this example, `-b` indicates that the input is a Boolean Program. Incidentally, we could have left out the `error:` bit because the label `E` occurs in only one function.

4.2 Experiments

This section reports on experiments carried out with Moped. The experiments were designed to test several aspects of the model-checker; the performance of both the reachability and the LTL algorithms, and the performance on different types of examples.

One of the motivations for studying pushdown systems instead of finite-state systems is the ability to handle the infinities caused by recursive procedure calls. Thus, a natural test of Moped's potential is to apply it to models of some recursive algorithms, e.g. Quicksort. Results for checking LTL properties on Quicksort and some other recursive algorithms are reported in Section 4.2.1.

While Moped is capable of verifying arbitrary LTL properties, Ball and Rajamani have presented an algorithm for checking reachability on Boolean Programs [3]. Since this algorithm is implemented in a tool called *Bebop*, it was naturally interesting to compare the performance of the two implementations (for a comparison on a more theoretical level, see Section 3.1.5). This was first done on a family of artificial examples which was proposed in [3]; the results of this comparison are in Section 4.2.2.

A series of more meaningful tests and comparisons were carried out on a large number of examples provided by Ball and Rajamani. These examples were automatically derived abstractions of C programs (more specifically, device drivers for the Windows operating system). These experiments are discussed in Section 4.2.3.

Finally, some conclusions drawn from the results of the experiments are discussed in Section 4.2.4.

4.2.1 Recursive algorithms

Since pushdown systems are strictly more expressive than finite-state systems, a natural application is to use them for problems which cannot be modelled by the latter, for instance programs with recursive procedures. In this section, this application is demonstrated by means of three examples: a sorting algorithm (Quicksort), a procedure which turns an array into a 'heap' (Heapify), and a procedure which checks whether two terms can be unified by variable substitution (Unify).

All three examples were modelled by hand as symbolic pushdown systems and tested for different sets of parameters (for instance, by varying the size of the array in Quicksort or Heapify). Varying these parameters allows to get an idea of what is possible with the verification methods, i.e. how large the examples are which can still be managed comfortably by the checker. For the Quicksort example, we also compared the performance of Moped with an exhaustive testing method.

We also used these examples to optimize the checker by testing some variants of the model-checking algorithms, for instance the question of backward and forward computation. This section mostly concentrates on reporting the final results of these optimization efforts; the impact of some of the options is discussed in Chapter 5.

Quicksort The objective of Quicksort is to take an array of integer values and sort it in ascending order. This is achieved by a divide-and-conquer strategy: Some element of the array is picked as the 'pivot', and all elements less or equal than the pivot are sorted to its 'left', all others to its 'right'. Then the array is split at the pivot and the problem is recursively solved on the two halves. Figure 4.4 shows an implementation attempt at this procedure. Two correctness properties which we might want to test for this implementation are

- whether the program sorts correctly;
- whether it terminates in all executions.

To test the termination property, we can actually abstract from the array contents and just observe the behaviour of the local variables. We then need to replace the decision if `a[hi] > piv` by non-deterministic choice (except in the first test, where we know that the outcome is false). A suitably abstracted program is shown in Figure 4.5, where $*$ denotes non-deterministic choice. Notice that the abstracted program overapproximates the possible executions of the original implementation, which means that correctness of the former implies the correctness of the latter.

We modelled the program from Figure 4.5 as a pushdown system, in which an additional procedure `main` makes an initial call to the `quicksort`

```
void quicksort (int left,int right)
{
     int lo,hi,piv;

     if (left >= right) return;
     piv = a[right]; lo = left; hi = right;
     while (lo <= hi) {
          if (a[hi] > piv) {
               hi--;
          } else {
               swap a[lo],a[hi];
               lo++;
          }
     }
     quicksort(left,hi);
     quicksort(lo,right);
}
```

Figure 4.4: Quicksort implementation, version 1.

```
void quicksort (int left,int right)
{
     int lo,hi;

     if (left >= right) return;
     lo = left; hi = right;
     while (lo <= hi) {
          if (hi == right || *) {
               lo++;
          } else {
               hi--;
          }
     }
     quicksort(left,hi);
     quicksort(lo,right);
}
```

Figure 4.5: Abstracted version of Quicksort.

```
local(qs0,qs1,qs2,qs3) int left(N), lo(N), hi(N), right(N);

(q <main0>)

q <main0> --> q <qs0 main1>
q <main1> --> q <main1>

q <qs0> --> q <> "reject"
         (left >= right)
q <qs0> --> q <qs1> "init"
         (left < right & left' = left & right' = right
                       & lo' = left & hi' = right)
q <qs1> --> q <qs1> "decrease hi"
         (lo <= hi & (hi != right | lo != left)
                    & lo' = lo & hi' = hi-1
                    & left' = left & right' = right)
q <qs1> --> q <qs1> "increase lo"
         (lo <= hi & lo' = lo+1 & hi' = hi
                   & left' = left & right' = right)
q <qs1> --> q <qs0 qs2> "left recursive call"
         (lo > hi & left' = left & right' = hi
                  & lo'' = lo & right'' = right)
q <qs2> --> q <qs0 qs3> "right recursive call"
         (left' = lo & right' = right)
q <qs3> --> q <>
```

Figure 4.6: Pushdown system corresponding to Figure 4.5.

procedure and waits for it to terminate. The values of `left` and `right` in the initial call are chosen non-deterministically. The resulting pushdown system is shown in Figure 4.6, where `N` is a symbolic constant which carries the number of bits used to represent each integer variable.

Checking termination in this model is equivalent to checking whether the initial call is completed in every run, i.e. whether `main1` is reached. When running Moped, it turns out that this is not the case because the program contains an error. Inspection of the counterexample printed by Moped reveals that the procedure can enter an infinite loop if the pivot element is the largest element in which case all tests for `a[hi] > piv` will be negative, and the first

```
void quicksort (int left,int right)
{
     int lo,eq,hi,piv;

     if (left >= right) return;
     piv = a[right]; lo = eq = left; hi = right;
     while (lo <= hi) {
          if (a[hi] > piv) {
               hi--;
          } else {
               swap a[lo],a[hi];
               if (a[lo] < piv) {
                    swap a[lo],a[eq];
                    eq++;
               }
               lo++;
          }
     }
     quicksort(left,eq-1);
     quicksort(lo,right);
}
```

Figure 4.7: Quicksort implementation, version 2.

recursive call has the same values for `left` and `right` as those with which the procedure was entered. (The author actually produced this faulty version by accident when he implemented the Quicksort algorithm in preparation for these experiments.)

One possible fix for the error is to keep tabs on the elements which are equal to the pivot and not to call the procedure recursively on them. Figure 4.7 shows a corrected program. Running Moped on the suitably corrected abstracted model now yields that all executions in the model terminate (and, since the model is an overapproximation, so do all executions of the program).

Having done this, we can try to verify the correctness of the sorting as well. Doing so requires to add the array contents of the model; moreover, after the system completes the initial call, we let execution branch to one of two labels, `ok` and `error`, depending on whether the result is correct or

K	N	M	globals	locals	Quicksort time	Quicksort memory	randomized time	randomized memory	testing time
1	5	20	20	26	4.6 s	13.2 M	4.7 s	13.3 M	3.4 s
1	5	30	30	26	18.2 s	21.4 M	20.0 s	22.1 M	4965.5 s
1	6	40	40	31	53.5 s	52.7 M	56.6 s	54.1 M	?
1	6	60	60	31	247.3 s	219.5 M	252.6 s	223.7 M	?
2	4	8	16	22	10.2 s	14.9 M	15.6 s	14.8 M	0.1 s
2	4	10	20	22	74.3 s	41.6 M	122.6 s	54.6 M	2.5 s
2	4	12	24	22	493.2 s	207.5 M	849.5 s	284.6 M	46.2 s
3	3	6	18	18	13.7 s	15.0 M	33.4 s	20.1 M	0.4 s
4	4	6	24	24	64.4 s	43.7 M	174.9 s	74.1 M	28.6 s
4	4	7	28	24	449.4 s	197.5 M	2807.4 s	507.0 M	546.4 s

Figure 4.8: Run-times and memory consumption for Quicksort example.

not. These two labels lead to infinite loops to ensure that every execution is an infinite run. (We omit printing the resulting pushdown system here, but it is included in the distribution of Moped.) The system depends on three parameters:

- `N`, the number of bits used to represent integer variables;
- `M`, the number of elements in the array (must be between 1 and $2^{\mathtt{N}} - 2$);
- `K`, the number of bits used to represent array elements.

In the model, the array contents are chosen non-deterministically before the initial call, thus every possible combination of array values is allowed. The formula `<>ok` now tests both the correctness of the sorting and the termination property. Running Moped reveals that the formula holds.

Figure 4.8 shows the running times and memory consumption for various values of the symbolic constants. These numbers were obtained by running the experiments on a Sun Ultra-60 machine with 1.5 GB of memory; the BDD library used by Moped was CUDD version 2.3.0 [38]. Moreover, the tool was instructed to use the *post** method to check the formula (compare Section 3.2.3). The columns 'global' and 'local' list the number of bits needed to represent the global and local variables, respectively.

The two columns beneath the word ‘randomized’ state the results for a variant of Quicksort in which the pivot element is not always chosen as the rightmost element, but picked randomly from the interval that is about to be sorted. (This decreases the expected time complexity of Quicksort and can be modelled conveniently by nondeterminism.) The column ‘testing’ shows the time needed for an exhaustive testing method which consists of a running a program which generates all possible inputs, runs version 2 of the Quicksort algorithm on all of them, and tests the result for correctness.

Looking at the numbers in Figure 4.8, it appears that the scalability of the model-checker lessens as `K`, the number of bits per integer, increases. This reflects the fact that the BDDs have to express increasingly complicated relations between several multi-bit integer variables; this is discussed in some more detail in Section 5.8. Still, for every fixed `K` the factor by which the run-time increases when `M` is increased is less than the factor by which the number of possible executions increases, thanks to the ability to store many states very compactly and hence the ability to compute many runs simultaneously. In other words, the model-checking method scales better than exhaustive testing does; indeed, for all the largest instances in the experiments the model-checker actually beats the time for exhaustive testing.

The bug in the first version which leads to non-termination can already be detected with small values for the symbolic constants, so that the error can be found by the checker almost immediately. The checker also produces a useful counterexample which aids in fixing the bug. In this sense, although the method cannot directly be used to show that the corrected algorithm works for arbitrary values of the parameters, it serves as an example for the usefulness of model-checking as a debugging technique. Incidentally, testing would be an inappropriate method to detect the error because it cannot distinguish between an execution that gets stuck and one that merely takes very long.

Heapify In this problem, given a tree whose nodes carry integer values, we want to convert the tree into a heap. A tree is a heap if each node is labelled with a value which is larger than the value of its children. This procedure is used, for instance, in sorting algorithms or priority queues.

For the Heapify problem, assume (w.l.o.g.) that the tree is binary. We simulate the tree by means of an array in which the children of entry number n are $2n{+}1$ and $2n{+}2$. The model involves three different recursive procedures:

				*pre** method		*post** method	
`K`	`M`	globals	locals	time	memory	time	memory
1	25	25	3	19.9 s	17.1 M	157.5 s	47.5 M
1	31	31	3	1239.9 s	311.4 M	6745.1 s	475.3 M
2	9	18	4	1.4 s	8.0 M	5.3 s	14.0 M
2	11	22	4	12.4 s	23.5 M	48.5 s	33.9 M
3	9	27	5	37.3 s	59.2 M	129.9 s	88.1 M
3	11	33	5	1938.1 s	858.4 M	5500.6 s	1198.3 M

Figure 4.9: Run-times and memory consumption for Heapify example.

- `hfy`, when called on node n brings the subtree below and including node n into heap form. It does so by first calling itself recursively on its child nodes (if any) and then brings node n to the right place by calling `adj`.
- `adj` compares node n with its children. If node n is already larger than its children, nothing needs to be done; otherwise the larger of the children swaps its place with n and the procedure branches into this child to restore its heap property.
- `chk` is called after the heapification process is complete and checks if the resulting tree is indeed a heap. If so, it pushes the stack symbol `ok`, otherwise `error`.

The complete model is not shown here; it is included in the Moped distribution. Both the correctness of the result and the fact that the procedure always terminates can be checked with the formula `<>ok`. Some experimental results are shown in Figure 4.9; there, `M` is the number of nodes in the tree (or array elements), and `K` is the number of bits available for the node labels.

The main observation of interest here is that this was the only example in these experiments in which the *pre** method has an advantage over the *post** method. In the other examples, *pre** compares much worse. The issue of *pre** versus *post** is discussed in greater detail in Section 5.5.

Unify Consider the class of terms which are made up of function symbols, variables, and constants. Given two terms t_1 and t_2, the problem of unification consists of deciding whether there is a substitution from variables into terms such that t_1 and t_2 are equal under this substitution, and if so, compute the substitution.

We can handle Unify in a similar fashion to Heapify: If we restrict the problem (w.l.o.g.) to functions with two arguments, then it can be portrayed as a binary tree and hence an array (albeit, admittedly, the modelling becomes awkward within the constructs offered by the input language of Moped). The model used in the experiments emulates the implementation from [24], which does not perform an 'occurs' check, i.e. when determining a term t as the substitution for x, it does not check whether x already occurs in t. The implementation maintains a pointer for each variable; this pointer is either a pointer to a subterm (i.e. an index into the array) or it is a 'chain' to another variable in case the two variables have the same substitution.

The model either pushes a `nouni` symbol if no unification is possible or an `ok` symbol when it completes the unification. We can then check the formula `<>(ok || nouni)` to test if the procedures terminates in every execution, or whether it can get stuck (the absence of non-termination is not immediately obvious, because the algorithm may have to recurse multiple times through subterms which have been bound to some variable, and because variable chains might form a loop). We cannot, however, test correctness in the sense of whether the algorithm always finds a substitution if one exists.

We first handled the case where we have two variables, one constant, and one function symbol. When the depth of the terms was limited to two, i.e. such that terms can have the form $f(g(a,x),h(b,y))$, Moped managed to verify the termination property in 70 seconds. The result was achieved using the *post** method and a slightly different variable ordering than in the other two examples (see the discussion on variable ordering in 5.8). However, even when just one term was allowed to have a depth of three, the verification discovered an error that leads to an infinite loop. The error happens because of the missing 'occurs' check. It took 276 seconds and 30 MB of memory to find this error.

When an 'occurs' check was added to the model, the termination property could be verified on the corrected model in 208 seconds. As an additional test, we then allowed terms to have three variables, two constants, and three function symbols. Checking termination for this larger set of terms still succeeded, but took 102 minutes and 520 MB of memory. For these settings the global variables in the model had 69 bits, and the local variables 16 bits.

4.2.2 A family of Boolean Programs

In the paper which introduced Boolean Programs and the Bebop checker [3], Ball and Rajamani demonstrated the performance of their checker on a family of constructed examples. These examples consist of n procedures, where n is a parameter. Despite being simple and quite artificial (see the details below), the examples prove some interesting points:

- Since no recursion is involved, the state space of the examples is finite. However, typical finite-state approaches would flatten the procedure call hierarchy, blowing up the program to an exponential size.

- The programs have exponentially many reachable states, yet reachability can be solved in time linear in n. This serves to demonstrate the usefulness of using pushdown systems in verification of programs: Not only are they strictly more expressive than finite-state systems, but they can exploit the modularity ingrained in a program's procedural structure.

- Finally, there are $\mathcal{O}(n)$ different variables in the program; however, only a constant number of them are in scope at any given time. For this reason the stack alphabet can be kept small, exploiting the locality of the variable scopes.

The examples consists of one `main` function and n functions called $\texttt{level}_i$, $1 \leq i \leq n$, for some $n > 0$. There is one global variable `g`. Function `main` calls $\texttt{level}_1$ twice. Every function $\texttt{level}_i$ checks `g`; if it is true, it increments a 3-bit counter to 7, otherwise it calls $\texttt{level}_{i+1}$ twice. Before returning, $\texttt{level}_i$ negates `g`. The checker is asked to find out if the labelled statement in `main` is reachable, i.e. if `g` can end with a value of false. Since `g` is not initialised, the checker has to consider both possibilities. Figure 4.10 shows a sketch of the program.

Running times (on the Sun Ultra-60) for different values of n are listed in the second column of the table in Figure 4.10. (In [3] results are reported for up to $n = 800$ where the running time is four and a half minutes using the CUDD package and one and a half minutes with the CMU package, but unfortunately the paper does not say on which machine. The author understands that current versions of Bebop may solve the problem a good deal more quickly.)

```
decl g;
void main()
begin
  level1();
  level1();
  if (!g) then
    reach:  skip;
  else
    skip;
  fi
end
```

```
void level_i()
begin
  decl a,b,c;
  if(g) then
    a,b,c := 0,0,0;
    while(!a|!b|!c) do
      if (!a) then
        a := 1;
      elsif (!b) then
        a,b := 0,1;
      elsif (!c) then
        a,b,c := 0,0,1;
      fi
    od
  else
    level_i+1(); level_i+1();
  fi
  g := !g;
end
```

n	time	'generous'
200	0.50 s	1.51 s
400	0.94 s	8.26 s
600	1.46 s	17.10 s
800	1.99 s	34.84 s
1000	2.41 s	56.81 s
2000	4.85 s	233.62 s
5000	13.63 s	1654.08 s

Figure 4.10: Boolean Program example from [3] and results.

More interesting is the question of space consumption. Moped has a peak number of 263 live BDD nodes, *independently of* n. On the contrary, Bebop's space consumption for BDDs increases linearly. The reason of this difference is that the BDDs involved require only four variables (one for the global variable g and three for the 3-bit counter of some procedure). Moped's default behaviour is to encode all local variables with the same BDD variables (see Section 5.9), therefore the very same BDDs nodes are used all the time.

The results in [3] show a running time which is quadratic in n (when it should be linear). The reason for this is that Bebop uses separate BDD variables to encode the local variables of all the functions and thus uses $\mathcal{O}(n)$ different BDD variables; however, no BDD has more than a constant number of variables. This triggers an inefficiency in some of the functions in the BDD packages (both CMU and CUDD), for instance the function for relabelling BDDs. It turns out that these functions take linear time in the number of BDD variables, which is $\mathcal{O}(n)$, even when the size of the BDDs is independent of n (so that one would expect the time consumption of the operations to be independent of n, too). Since $\mathcal{O}(n)$ such operations are required to solve the reachability query, the overall time consumption becomes quadratic. It turns out that if Moped is instructed to use separate BDD variables for the locals like Bebop does, the same problem occurs (unsurprisingly, since Moped also uses CUDD). The results with this encoding are listed in Figure 4.10 in the column marked 'generous'. Likewise, the authors of Bebop report that when Bebop was instructed to use Moped's encoding, its running time was reduced to linear. The question of how the allocation of program variables to BDD variables affects the model-checking times is discussed in more detail in Section 5.9.

4.2.3 Abstractions of C programs

The performance of Moped was tested on a large number of examples provided by the authors of SLAM [4], Tom Ball and Sriram Rajamani. These examples were created as abstractions of C programs by the SLAM toolkit. The aim of SLAM is to verify temporal safety properties (roughly speaking, the class of properties which express that 'something bad does not happen') on C programs. Since this problem is undecidable in general, the process may not terminate; however, it has been successfully used to validate correct interface usage in device drivers. We briefly explain how the process works:

Given a C Program P and a temporal safety property φ, the toolkit first synchronizes P with an automaton for φ to end up with a program P'. This construction has the property that the original program P satisfies φ if and only if a certain label L is reachable in P'. The latter is then checked by a process of iterative refinement which consists of three steps:

1. In the i-th iteration, the program P' is first abstracted into a Boolean Program B_i which safely overapproximates the possible behaviours of P'. The abstraction is made with respect to a set of predicates E_i which occur as boolean variables in B_i and which express properties of P'. For instance, if E_i contains the predicate $x > 0$, then B_i would contain the variable $\{\mathtt{x} > \mathtt{0}\}$, and the instructions in B_i would reproduce the effect which their counterparts in P' have on the validity of the predicate (non-determinism is used if it turns out to be impossible whether some instruction validates or invalidates a predicate, hence the characterization of B_i as an overapproximation). The more predicates E_i contains, the better the approximation gets. An initial set of predicates E_1 is derived from φ; in subsequent iterations, the set of predicates is extended by the results of step 3. The abstraction step is implemented by a program called c2bp.

2. In the second step, the program Bebop checks whether L is reachable in B_i. If L turns out to be unreachable, we can conclude that L is also unreachable in P' (because B_i is an overapproximation) and hence φ holds.

3. If in the previous step L is reachable through some path p, one has to check whether p is also a feasible path in P' or whether it it 'spurious', i.e. whether it is just due to the overapproximation. This is determined by a program called Newton. If Newton determines that the path is feasible, then φ does not hold and p is a counterexample. If p is spurious, then Newton finds an 'explanation' for this and extends the set of predicates in such a way that the path becomes impossible in subsequent iterations.

In this context, Moped can be used to implement step 2. The examples used in the experiments came from four different groups:

- a regression test suite of 64 small programs designed to test various features of the C language;

- four Windows NT device drivers which were checked for correct lock acquisition/release sequences;
- one driver which was checked for conformance to a state machine for I/O request packet completion;
- a group of large Boolean Programs derived from a serial driver.

In every single example, Bebop and Moped agreed on the result of the reachability analysis, i.e. whether the label in question was reachable or not, which gives the authors of both tools some confidence in the correctness of their checkers. In all the tests, Moped performed better than Bebop by a significant amount. By comparing several aspects of the implementations of both checkers, we were able to identify some of the reasons for this. Moreover, Moped's output was made compatible with that of Bebop so that it became possible to run the SLAM process with Moped in Bebop's place.

The following paragraphs provide more details on the findings. All running times were measured on a machine with a 800 Mhz Pentium-3 CPU and 512 MB of RAM. Unless noted otherwise, the checkers were run with the default options and with error trace generation for the Newton tool, i.e. the way they would be used in a SLAM process.

Regression test suite The test suite consisted of 64 small C programs (between 30 and 150 lines of code) designed to test the ability of the SLAM tools to cope with language features such as structure fields, pointer dereferences, dynamic memory allocation and others. Running these examples through SLAM resulted in 178 iterations in total. Figure 4.11 shows various statistics of these experiments. Average and maximum numbers are computed over all 178 runs.

The running time is the sum of all 178 running times (in seconds). The numbers show that Moped is faster by a factor of more than 4. The figures on memory consumption are taken from statistics provided by the CUDD package (which both checkers use). The number "memory in use" is given in megabytes, the numbers of BDD nodes in thousands. An interesting pattern emerges: Moped uses less memory and has a lower peak number of BDD nodes, but in fact it has a higher peak number of *live* BDD nodes. Normally, this should cause a higher overall memory consumption for Moped. It is not clear what causes this oddity, in particular because both checkers use CUDD's standard parameters for garbage collection and other activities relevant to

	Bebop		Moped	
total running time	209.7		46.4	
memory in use avg/max	4.4	9.0	4.3	6.8
peak BDD nodes avg/max	12.1	273	7.1	145
peak live BDD nodes avg/max	1.8	40	4.3	105
BDD variables avg/max	65.9	465	59.6	362

	average	maximum
global variables	1.3	10
max local variables	6.6	36
max arguments	2.4	8
max return values	1.7	6

Figure 4.11: Statistics for regression test suite.

memory consumption. Finally, we list the number of different BDD variables managed by CUDD. The numbers here differ because the two checkers have different strategies for relating BDD variables to program variables (see the discussion of this in Section 5.9).

The second part of the table gives more details about the nature of the Boolean Programs which were checked. It lists information about the numbers of global (boolean) program variables, and the maximum numbers of local variables. The latter includes formal parameters, local variables declared at the beginning of the function and those declared 'on the fly'. These two factors determine how many BDD variables will occur in the BDDs. The last two lines concern the maximum number of arguments and the maximum number of return values.[3] These two numbers are interesting because calls to functions with arguments, or return statements with return values correspond to parallel assignments between groups of BDD variables. Therefore, calls and returns with many arguments can easily translate to huge BDDs and render the checking process practically infeasible if the variable ordering is inappropriate.

[3]This means that for each program, we first determined the maximum number of local variables/arguments/return values over all functions; then we computed the average and maximum of those numbers over all 178 examples.

	Bebop		Moped	
total running time	944.2		88.5	
memory in use avg/max	10.7	13.1	7.6	12.8
peak BDD nodes avg/max	286.8	437	161.7	470
peak live BDD nodes avg/max	67.6	169	100.5	267
BDD variables avg/max	1894.1	2289	864.2	1028

	average	maximum
global variables	5.5	16
max local variables	40.5	58
max arguments	9.9	12
max return values	12.0	15

Figure 4.12: Statistics for the four drivers (lock/unlock property).

Locking/unlocking sequences The second set of Boolean Programs was derived from four Windows NT drivers. These drivers need to acquire and release so-called 'spin locks' from the kernel. It is an error to acquire a lock twice in a row without releasing it, or to release it when it is not acquired. Four drivers with between 2200 and 7600 lines of code were checked for correct lock/unlock sequences. Checking the drivers took 32 iterations in total (between 1 and 16 iterations per driver). The results of running both checkers on the 32 resulting Boolean Programs are listed in Figure 4.12. Moped was faster by a factor of more than 10. The same pattern as before emerges in the memory usage; Moped uses less memory despite using more live BDD nodes.

Request packet completion When the operating system wants a driver to perform a specific operation it invokes a certain 'dispatch routine' of the driver and passes to it information about the request. The driver must either process the request immediately or may queue it for later processing. In both cases the driver must adhere to a certain protocol which involves, e.g., calling certain kernel functions in the right order (the details are out of scope here). The protocol can be described with a state machine and be formulated as a safety property. One of the drivers from the previous group of examples (with 4855 lines of code) was checked for conformance with this property, which took 16 iterations. Figure 4.13 lists the results of the checking, which

	Bebop		Moped	
total running time	7615.1		142.4	
memory in use avg/max	24.9	32.1	14.9	21.9
peak BDD nodes avg/max	928.5	1368	551.4	956
peak live BDD nodes avg/max	685.4	1126	330.2	841
BDD variables avg/max	3962.4	4077	1707.9	1765
	average		maximum	
global variables	17.1		18	
max local variables	45.5		48	
max arguments	18.0		18	
max return values	18.0		18	

Figure 4.13: Statistics for iscsiprt driver (request packet completion).

are similar in nature to the previous results. It should be noted that part of Moped's advantage in running time is due to the generation of error traces. Bebop tries to generate the shortest error trace whereas Moped just generates *some* error trace (not necessarily the shortest). When the checkers were run without error trace generation, Bebop's time was reduced to about 30% of the time quoted in Figure 4.13.

Large examples The four largest Boolean Programs on which we conducted experiments were derived from a serial driver (with almost 27000 lines of code). These presented potentially high challenges because some of the functions in the Boolean Programs had large numbers of arguments and return values (see Figure 4.14). Nevertheless, Moped managed to find error traces in all of them in times between 11 and 14 seconds each. Again, it should be noted that Bebop is handicapped by its slow error trace generation. Generating the traces alone turned out to account for about 2500 seconds alone, which is probably just due to an oversight in the implementation but not to a fundamental difference between the checkers.

Computing the whole state space In the examples above, most of the time the outcome of the reachability analysis was positive due to the nature of the SLAM process, which rules out spurious error traces until a real error or no error is found. Finding an error trace, however, can often be done by

	Bebop		Moped	
total running time	3488.1		50.7	
memory in use avg/max	24.4	25.9	23.2	25.5
peak BDD nodes avg/max	601.6	682	489.8	632
peak live BDD nodes avg/max	143.8	225	231.0	265
BDD variables avg/max	10439.2	10461	7049.0	7070

	average	maximum
global variables	28.0	28
max local variables	325.6	334
max arguments	22.0	22
max return values	27.0	27

Figure 4.14: Statistics for serial driver.

exploring only a small part of the reachable state space. Out of curiosity, we also explored how long it took to compute all reachable states: For the four drivers and the lock/unlock properties, the 32 programs took just 94 seconds (hardly more than before), for the request packet completion property it took 1996 seconds for all 16 programs (14 times more), and for the four serial driver examples it took 157 seconds (about 4 times more).

Trace generation While Bebop and Moped always came to the same conclusion about whether the label in question was reachable or not, they gave different error traces. Bebop always generates the shortest error trace, whereas Moped's default behaviour does not have such a guarantee. When experimenting with the four drivers (32 programs), we found that Moped's error traces were, on average, about one third longer than Bebop's; in the largest single deviation Moped's trace was three times as long. The trade-off between higher running-time and shorter trace generation is discussed in more detail in Section 5.4.

Running Moped inside SLAM In the results reported above, both checkers were run on the same set of Boolean Programs so that the results are directly comparable. These programs were obtained by first running the SLAM process with Bebop, then taking the resulting programs and checking them a second time with Moped.

An obviously interesting experiment was to plug Moped into the SLAM toolkit in Bebop's place, and to see whether Moped's different trace generation would affect the process. We first tried this out with the regression suite. All 64 examples could be verified; for three examples the process took one iteration less with Moped than it had with Bebop; for six examples it took one iteration more, and for one example it took two iterations more. The effect on the overall running time was negligible, however, because the examples were so small. Including the time for running the other SLAM tools, running SLAM on all examples of the regression suite took about 5 minutes and 40 seconds.

Unfortunately, no conclusive data could be gained from trying this process with the drivers. In one driver, the initial iteration showed that the label was not reachable, so no trace was created. In another driver, the number of iterations increased from two to four, but the overall running time of the process still decreased because Moped solved the four reachability problems quickly enough. In the other two drivers and in the check for request packet completion, Moped's error traces caused Newton to report a potential null pointer dereference in the driver, which ended the verification process. In the limited time available for carrying out the experiments, this issue could not be resolved.

Differences When comparing the algorithms on which the reachability analysis of the two checkers is based, it turns out that they are in fact very similar. This point is argued more precisely in Section 3.1.5. Also, both use the same BDD library and the same method for determining the variable ordering (which Moped in fact borrows from Bebop, see also Section 5.8). However, the differences in running time are considerable, and a discussion of implementation details with the authors of Bebop led to insights about some of the reasons:

- Part of the reachability analysis consists of computing the effects of functions (i.e. the relation between arguments and return values) which must be 'fed back' into the control flow of the callers. This can be done in two modes, 'eager' and 'lazy' (see Section 3.1.5). Moped uses lazy evaluation which saves some time. A comparison to an 'eager' version of Moped is carried out in Section 5.6.

- The checkers use different strategies for relating program variables to BDD variables; in general, Moped uses less BDD variables. Because some BDD operations take time proportional to the number of BDD variables in the manager (see Section 4.2.2), this accounts for some of the savings; see also Section 5.9.

- Bebop generates shortest error traces, whereas Moped's default behaviour is to generate the first trace it finds in depth-first search. However, the shortest trace generation is bought at a higher running time. This is discussed in more detail in Section 5.3.

Possible improvements There is still plenty of room to improve the performance. Some ideas are listed below:

- The nature of the SLAM process means that quite often we have a couple of 'yes' instances (in which the label is reachable) before one 'no' instance. Therefore, the process can be sped up a lot by optimizing the algorithms towards finding the error trace as quickly as possible. Finding an error trace often requires exploration of only a small part of the state space. In fact, in quite a few examples most of the time is spent building the BDDs for the transition relations whereas the reachability analysis itself is very quick. Thus, a natural improvement would be to build the transition relations only 'on demand', i.e. only when the corresponding statement is actually reached during the search.

- For the same reasons, the search could benefit from being 'smarter'. Currently both checkers simply examine every transition they find until they happen upon the error label. It would be interesting to try out techniques which explore the control flow of the program first and identify 'promising' paths which are then treated preferentially in the search.

- A large part of the statements in the Boolean Program examples are actually *skip* statements. Many other statements are parallel assignments with just a few variables. The BDDs which express the transition relations for these statements change just the left-hand side variables and copy the pre-values into the post-values for all other variables. When reachability is propagated over these statements, the accompanying BDD operation is a join (with existential quantification) and a relabelling, both over a full set of global and local variables. Instead, one could represent the transition relation for these statements with

BDDs which express just the variables which are changed, and perform the existential quantification and relabelling over just these variables. In the case of *skip* statements, this would amount to simply copying the BDD of reachable valuations from the program point before the statement to the program point after it.

- Both checkers can perform some optimizations on the Boolean Programs which lead to smaller BDD sizes (e.g. by identifying dead variables). However, in both checkers, these techniques interfere with their error trace generation. However, it appears that the two techniques can be combined with a little extra work, see Section 5.7.

4.2.4 Conclusions

The experiments conducted in the previous sections show that the model-checking methods are efficient enough to succeed on systems with a couple of dozens or even a few hundreds of binary variables. The recursive algorithms verified in Section 4.2.1 could not have been handled with finite-state methods. In the Boolean Program examples, the ability to handle recursion was of lesser importance, but pushdown systems proved to be a convenient and natural model for handling procedures. As the examples from Section 4.2.2 show, they can be vastly more efficient than finite-state methods even when recursion is ruled out.

To the author's knowledge, Moped is currently the only model-checker which can handle LTL efficiently for large-scale pushdown systems. The only other implementation which could theoretically be used for this purpose is *Alfred* by Daniel Polanský [35], which is a model-checker for the alternation-free mu-calculus. However, this checker may have exponential run-time even for some formulas of LTL, and it has no support for symbolic representations. For reachability, which is a subset of LTL, an alternative implementation exists in Bebop. The results show that Moped performs clearly better than Bebop on reachability properties. We have identified some differences in the algorithms which account for part of the performance gap, and have listed some ideas which can increase the running times even further.

The recursive algorithms from Section 4.2.1 and the abstractions of C programs from Section 4.2.3 have quite different characteristics; for instance, the latter have a much larger control flow and involve far more BDD variables. However, they appear to constitute easier problems. While this may seem surprising at first glance, there are in fact good reasons why this should be so. Control flow only contributes a linear factor to the complexity of the algorithms, and BDD operations contribute by far the largest part to the overall running time and memory consumtion. In the recursive algorithms (Quicksort, Heapify, Unify), the BDDs have to express quite complicated relationships between several integer variables (e.g. in Quicksort we swap two entries in the array whose indices are given by other variables) and there is little redundancy between them (e.g. in the array sorted by Quicksort the array entries have no relation to each other, except that in the end they are sorted). In contrast, the Boolean Program examples often express less complicated relations. Many instructions are simply the *skip* statement, others are parallel assignments in which the right-hand sides typically have simple forms. Also, the boolean variables express predicates, some of which are contradictory (e.g. {`x=0`}, {`x=1`}, {`x=2`}), which places additional restrictions on the reachable state space. Moreover, the properties which we checked for the recursive algorithms were true, but to verify this Moped had to search the entire reachable state space. In the Boolean Program examples, most of the time the outcome of the reachability analysis was positive, so the search could be aborted before the entire reachable state space was computed.

The results presented in this chapter were achieved through experimenting with a number of different settings and options in the checker. We could establish a set of settings which were superior for most examples and which were therefore made the default in Moped. These settings are discussed in detail in Chapter 5.

Chapter 5

Implementation aspects

Chapter 3 analyzes the model-checking algorithms from an asymptotic point of view. However, it is well-known that asymptotic complexity often tells little about how algorithms perform in practice, and in an actual implementation the programmer has many design choices and fine-tuning options for special cases which can have a drastic effect on their efficiency. During the implementation of Moped efforts have been made to explore some of these options. The experimental results listed in Chapter 4 are the result of this exploration; in the following, some of the design choices and their effects are discussed.

5.1 A data structure for transition tables

The algorithms for pre^* and $post^*$ are at the heart of the model-checker. We discuss data structures for pre^* here as it makes the presentation slightly easier, but it is straightforward to transfer the ideas to $post^*$. Section 3.1.3 discussed data structures only as far as was necessary to prove the complexity results; here we provide some more details. Studying Algorithm 1 (for pre^*, reproduced in Figure 5.1 for easier reference) reveals that our data structures must provide for the following operations:

1. Given a pair q, γ, we want to traverse all rules with q, γ on the right-hand side (lines 7 and 9).

2. The set *trans* contains the automaton transitions which still need to be processed. The operations needed for *trans* are adding a transition

Algorithm 1

1 $\mathit{rel} := \emptyset;\ \mathit{trans} := (\rightarrow_0);\ \Delta' := \emptyset;$
2 **for all** $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $\mathit{trans} := \mathit{trans} \cup \{(p, \gamma, p')\};$
3 **while** $\mathit{trans} \neq \emptyset$ **do**
4 pop $t = (q, \gamma, q')$ from *trans*;
5 **if** $t \notin \mathit{rel}$ **then**
6 $\mathit{rel} := \mathit{rel} \cup \{t\};$
7 **for all** $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in (\Delta \cup \Delta')$ **do**
8 $\mathit{trans} := \mathit{trans} \cup \{(p_1, \gamma_1, q')\};$
9 **for all** $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta$ **do**
10 $\Delta' := \Delta' \cup \{\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q', \gamma_2 \rangle\};$
11 **for all** $(q', \gamma_2, q'') \in \mathit{rel}$ **do**
12 $\mathit{trans} := \mathit{trans} \cup \{(p_1, \gamma_1, q'')\};$
13 **return** *rel*

Figure 5.1: The pre^* algorithm (same as Figure 3.3).

(lines 2, 8, 12), emptiness check (line 3), and finding and removing a transition (line 4). Also, we want membership test to avoid duplicate entries; imagine that all additions to *trans* are preceded by a membership test.

3. The set *rel* contains all transitions which have been processed. We need operations for membership test (line 5) and adding a transition (line 6). Moreover, given a pair q', γ_2 (where γ_2 occurs in a rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle$), we want to find all transitions of the form (q', γ_2, q'') (line 11).

As far as item 1 goes, we already suggested in Section 3.1.3 that all rules of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' w \rangle,\ \ w \in \Gamma \cup \{\varepsilon\}$, be put into a bucket labelled by (p', γ'), and to organise the buckets in a hash table. If we add rules to Δ' (line 10), we can simply drop the new rule into the appropriate bucket in constant time.

Also in Section 3.1.3 we suggested that *rel* be implemented as a hash table, and *trans* as a stack. To avoid duplicate entries in *trans*, we also suggested to keep a second hash table which holds the elements of *trans*. Actually, we can get away without all these duplicities. We just need one hash table with an integrated stack. This is done as follows: The hash table stores the set $rel \cup trans$. To each transition t in the hash table we associate a pointer p_t. If t is in *trans*, then p_t points to the next element in the stack, or *nil* if it is the last element. We keep a special pointer named *stack* which points to the first element in the stack. It is now clear that adding a transition, emptiness check, and finding and removing a transition can be done in constant time.[1] Because we never want to process any transition twice, we can actually replace membership test in *trans* by membership test in $rel \cup trans$ and implement it with a hash table lookup. This also eliminates the need for the test in line 5.

Lines 4 to 6 are now implemented as follows: We take t to be the transition pointed to by *stack*, then set $stack := p_t$. Now we are free to use p_t for other means. The only operation we still lack is to find the transitions of the form (q', γ_2, q'') in *rel* for a given pair q', γ_2. Our solution is to keep a linked list for each pair, e.g. to have a pointer p_{q',γ_2} which points to the first element in the list. Assume that $t = (q', \gamma_2, q'')$ for some q''. As we 'move' t from *trans* to *rel*, we simply set $p_t := p_{q',\gamma_2}$ and have p_{q',γ_2} point to t (in this order). Now line 11 can be implemented by simply traversing the list of transitions starting at p_{q',γ_2}. The pointers to the start of the linked lists can be either initialized to *nil* at the beginning of the algorithm or be created on demand and kept in a hash table.

5.2 Symbolic cycle detection

In Section 3.2 we pointed out that model-checking LTL on pushdown systems requires finding a set of so-called repeating heads, which in turn reduces to identifying strongly connected components with a 1-labelled edge (henceforth called *1-components*) in the head reachability graph. However, the classical algorithm for finding these components by Tarjan [40] is not suitable for graphs with a symbolical representation. The reason is that Tarjan's algorithm uses depth-first search which considers each node individually whereas the strength of symbolic representations lies in dealing with potentially very large sets of states at a time. Symbolic algorithms for finding the 1-components have worse complexity bounds than depth-first search, but in practice they perform much better on very large graphs.

[1] Actually we can also implement a LIFO queue here, or even a priority queue (albeit possibly at a cost).

Symbolic cycle detection is a well-known problem for which several solutions have been proposed (see, for instance, the comparative surveys in [23, 36]). So far, no algorithm is universally regarded as the best; some have incomparable complexity estimations depending on different characteristics of the graphs. They are thus heuristics each of which can outperform the others on certain examples. For the experiments we picked some algorithms which are representative of different classes of heuristics (based on their performance in [23, 36]). We also considered how the particular nature of our examples could be used to speed up the process. This led to two optimizations: partitioning and a preprocessing step.

Partitioning The head reachability graph has a partitioned edge relation. If the control locations have the form $P_0 \times G$ and the stack alphabet the form $\Gamma_0 \times L$, the graph is represented by a set of symbolic edges $(p, \gamma) \xrightarrow[{[R]}]{b} (p', \gamma')$ where $p, p' \in P_0$, $\gamma, \gamma' \in \Gamma_0$, $b \in \{0, 1\}$ and $R \subseteq (G \times L) \times (G \times L)$; if $(g, l, g', l') \in R$, then $((p, g), (\gamma, l)) \xrightarrow{b} ((p', g'), (\gamma', l'))$ is an edge in the graph.

We could also encode b into R, but keeping it explicit allows us to rule out parts of the graph which are guaranteed not to have 1-components: First, we obtain a simplified, explicit graph by dropping the relations R from the symbolic edges (in terms of program analysis, this leaves us with the control flow and Büchi state information). The resulting graph is an overapproximation, but we can conveniently use Tarjan's algorithm on it to identify *potential* 1-components. We can then narrow down the symbolic search to these potential 1-components.

Instead of Tarjan's algorithm, finite-state model-checkers often use a more memory-efficient method by Courcoubetis et al [14] to identify cycles. However, this method is not useful for our purposes at this point because it stops at the first cycle it finds and in particular does not identify the components, but only individual cycles.

Example We consider the Quicksort example once more. In our experiments, we modelled the algorithm with the system in Figure 4.6. The formula $\Diamond ok$ was used to check that every execution reaches the label *ok*, which signifies that the sorting procedure has terminated and the result is sorted correctly.

The left half of Figure 5.2 shows the Büchi automaton for the negation

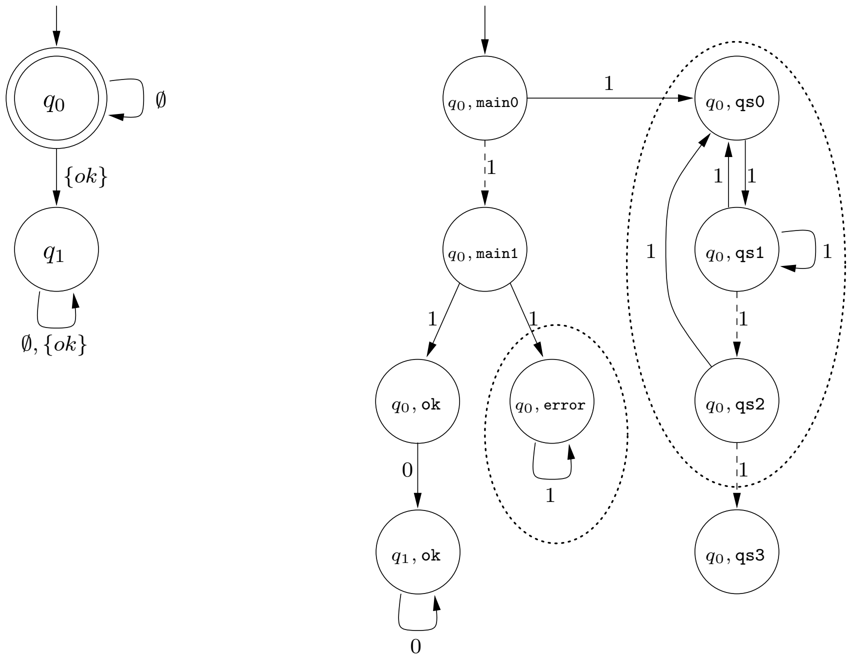

Figure 5.2: Left: Büchi automaton for $\neg(\Diamond ok)$. Right: Simplified graph for the product of the Büchi automaton and the Quicksort example.

of this formula. The model-checking procedure synchronizes this Büchi automaton with the pushdown system and derives the head reachability graph from the synchronized system. The right half of the figure shows the reachable part of the simplified graph obtained in this case (see Definitions 3.6 and 3.8 for details on how the graph is constructed). Potential 1-components are indicated by dotted circles.

In the following we assume that we have already identified a potential 1-component and that we restrict our symbolic search efforts to the edges within this component. Moped implements a number of different methods for the symbolic search and a preprocessing step, which can be combined with any of the methods.

Preprocessing The preprocessing step identifies and removes *trivially irreversible transitions*. In a symbolic edge

$$(p, \gamma) \xrightarrow[{[R]}]{b} (p', \gamma')$$

the set R is a relation between the pre-values and the post-values of tuples of variables. We say that R contains a trivially irreversible transition if a variable can increase in R, but cannot decrease in any transition in the potential 1-component that (p, γ) and (p', γ') belong to (or vice versa). Such transitions cannot form part of a cycle and can be removed.

Checking whether a variable can increase or decrease is easy; for a boolean variable whose pre- and post-values are represented by the BDD variables v and v', respectively, we merely quantify away all BDD variables other than v and v' and inspect the resulting (small) BDD. For integer variables, which are represented by multiple BDD variables, we first perform this kind of check for the BDD variable which represents the most significant bit (and remove the transitions found to be trivially irreversible), then repeat the procedure in descending order of significance.

Methods The following paragraphs briefly describe the four methods available for detecting cycles. Except for the first method, they work by computing sets of states in the head reachability graph $\mathcal{G}$ restricted to the potential 1-component currently under consideration.

- **Transitive closure** Conceptually the simplest method, this computes the transitive closure of the transition relation, i.e. $\rightarrow^*$. However, $\rightarrow^*$ is potentially very large and the operations to compute it, though few, are expensive, usually making this an inefficient method.

- **Xie-Beerel method** [43] This is basically a divide-and-conquer strategy for identifying components in symbolically represented graphs. It is based on the observation that if we pick any node n of the graph, $\mathit{post}^*(\{n\}) \cap \mathit{pre}^*(\{n\})$ is an SCC. If two nodes of this set are connected by a 1-labelled edge, we have found a 1-component. Otherwise, the graph is subdivided into $\mathit{post}^*(\{n\}) \setminus \mathit{pre}^*(\{n\})$ and $(P \times \Gamma) \setminus \mathit{post}^*(\{n\})$ and the search continues recursively on these smaller graphs.

- **Emerson-Lei method** [17] Often considered the standard method for symbolic cycle-detection, this algorithm computes the set of nodes from which 1-components can be reached (the set is empty if no such components exist). Thus, it gives us an overapproximation of the real set of repeating heads Rep. However, it still suits our purposes: in the pre^* method, where we compute $\mathit{pre}^*(\mathit{Rep}\,\Gamma^*)$, the 'extra' states would be reached in the *pre** step anyway, and in the *post** method, where we only consider the reachable heads, it is enough to know that a repeating head exists.

- **OWCTY method** [23] Fisler et al proposed a variant of the Emerson-Lei method which they call OWCTY (one-way catch-them-young). The method computes the same set of nodes as Emerson-Lei but in a slightly different fashion. Extensive experiments carried out in [23] suggest that, although the Emerson-Lei method can outperform OWCTY on certain examples, OWCTY performs better on a wider variety of examples.

Experiments We ran the methods with and without preprocessing on various instances of the Quicksort, Heapify, and Unify examples from Section 4.2.1. The running times in Figure 5.3 list the times it took to compute the repeating heads only. Except for varying the method for computing the repeating heads, all settings were the same as in Section 4.2.1. To make the comparison more interesting, the instances we chose were those with the largest running times. In Quicksort, we used the randomized variant; in Unify, we ran the same tests as in Section 4.2.1, i.e. the first two runs were on the faulty version without 'occurs' check where term depth two failed and term depth three succeeded in finding an error, and the other two runs were on the corrected version with 'occurs' check with a small number of variables, constants and functions (2/1/1) and with a larger number (3/2/3).

As far as a comparison of the four cycle-detection methods is concerned, the transitive closure method turned out not to be competitive (as could be expected), in most cases failing to complete the tasks within an hour. The other three algorithms performed about equally well in these examples. Therefore, we show the time for just one of them (OWCTY) and compare its performance with and without preprocessing.

The preprocessing manages to significantly reduce the time in Quicksort and Heapify. A closer look at the examples reveals how the preprocessing affects the computation times:

- In the Quicksort example most time goes into checking the potential 1-component which includes $(q_0, \texttt{qs0})$, $(q_0, \texttt{qs1})$, and $(q_0, \texttt{qs2})$. Here the preprocessing step detects that the variable `left` cannot decrease and removes the steps which increase it (compare the pushdown system

			with	w/o
			preprocessing	
`K`	`N`	`M`	Quicksort	
1	6	60	17.8 s	339.2 s
2	4	12	68.3 s	630.7 s
3	3	6	3.1 s	13.1 s
4	4	7	59.3 s	1065.4 s
`K`		`M`	Heapify	
1		31	0.03 s	0.08 s
3		11	0.29 s	0.71 s

	with	w/o
	preprocessing	
Unify w/o occurs		
depth 2	3.3 s	3.3 s
depth 3	24.8 s	24.9 s
Unify with occurs		
small (2/1/1)	11.2 s	11.1 s
large (3/2/3)	472.7 s	470.1 s

Figure 5.3: Times for computing repeating heads (OWCTY method).

in Figure 4.6). Similarly, the steps which decrease `right` are removed because `right` cannot increase.

- The Heapify example models a binary tree with an array. Here the preprocessing step recognizes that the recursive procedures which manipulate the tree always move downwards in the tree so that the variable which holds the index of the array element to be processed never decreases.

- The preprocessing step is not effective in the Unify example. In principle, Unify is similar to Heapify: it also models two binary trees with an array and recursively descends through the trees. However, the monotonicity of the recursive descent is broken when the algorithm has to descend through subtrees which were previously fixed as the substitution of some variable.

The preprocessing step speeds up the methods for finding cycles because it removes transitions which are guaranteed to be outside of cycles, thus reducing the diameter of the graph, which is a factor in the complexity of all four methods. It seems reasonable to expect that the preprocessing could be beneficial in other examples with recursion or loops as well. Typically, recursive procedures or loops exploit some kind of monotonicity property to guarantee termination, e.g., if $x_1, \ldots, x_n$ are variables, then there is some function g such that

- the value of $g(x_1, \ldots, x_n)$ never increases;
- at every point during execution of the procedure/loop, $g(x_1, \ldots, x_n)$ must decrease eventually;
- termination is implied if $g(x_1, \ldots, x_n)$ goes below a certain value.

The preprocessing step can check the first condition if the monotonicity of g somehow manifests itself in the monotonicity of single variables, as it does in Quicksort or Heapify. If this is not the case, a user who has an idea of how g might look like could 'help' the model-checker by introducing an additional variable y and keeping the condition $y := g(x_1, \ldots, x_n)$ invariant (albeit possibly at the cost of increasing the time needed by the $pre^*/post^*$ steps). There seems to be little reason not to use the preprocessing step; even if it does not remove any transitions, it does not take a lot of time.

The examples cannot serve as a basis for a proper comparison between Xie-Beerel, Emerson-Lei, and OWCTY. One reason is that we only test them on *weak* systems [28], i.e. systems where the SCCs contain either only 1-edges or only 0-edges. Tests on a wider variety of systems (like in [23]) would be in order here to expose the differences.

To conclude, regardless of which method is employed for cycle detection (except for transitive closure), it seems promising to use the preprocessing step which removes the trivially irreversible transitions. Naturally, Moped could also profit from future advances in symbolic cycle detection. Another possible improvement would be to use an adaptive approach: once a potential 1-component has been identified, one could choose a specialized cycle detection method based on an analysis of the structure of the component. Specialized algorithms exist, for instance, for so-called weak and terminal systems [7] which would apply for the examples we considered here.

5.3 Symbolic trace generation

The usefulness of model-checking for debugging purposes is greatly enhanced if the model-checker is able to print an execution which exhibits a violation of the property if one is found. The $pre^*/post^*$ algorithms compute the predecessors/successors of a set of of configurations as the language accepted by some finite automaton. Section 3.1.6 shows how the path from a predecessor/successor of a set C back to some configuration of C can be reconstructed by annotating the transitions of these automata with additional information. Computing and storing the annotations does not increase the asymptotic complexity of the algorithms. Here, we discuss practical issues of reconstructing paths and generating the annotation for the case of symbolic systems.

In Section 3.1.6, the annotation to the transitions in the finite automaton contains the pushdown rule that was used to create it (plus an automaton state if the transition causing the addition was a 'composite' transition involving an ε-transition; here we concentrate on the rule only to keep matters simple). For normal (non-symbolic) pushdown systems, this translates to annotating each transition with a pointer to a memory structure containing some rule. For symbolic systems, it translates to expanding the BDDs which represent the transition relations. For instance, assume that we have a symbolic rule $r := \langle p, \gamma\rangle \underset{[R_r]}{\hookrightarrow} \langle p', \gamma'\rangle$. When computing $post^*$, if we have $p \xrightarrow[{[R]}]{\gamma}{}^{*}\, q$ for some state q, we would normally add $p' \xrightarrow[{[R']}]{\gamma'} q$, where

$$R' = \{\, (g', l', g'') \mid \exists g, l\colon (g, l, g', l') \in R_r,\ (g, l, g'') \in R \,\}$$

When generating the annotation, we have to account for the fact that the tuples in R' may have been caused by different rules. Therefore the annotation becomes a list of items (r_i, R_i) where the tuples in the sets R_i are a partition of R' and such that r_i caused the addition of the tuples in R_i. Moreover, if we transfer the path reconstruction method verbatim, then we would have to extend the relation in order to record the 'origin' of each tuple, e.g. the addition in the example above would cause a list item (r, R'') where

$$R'' = \{\, ((g', l', g''), (g, l)) \mid (g, l, g', l') \in R_r,\ (g, l, g'') \in R' \,\}.$$

Let us call this the *complete* method for reconstructing paths. Obviously, R'' may have a rather larger representation than R', and there is a penalty in both time and memory for handling it. Instead, Moped uses an *incomplete* method which eschews the computation of R'' during the reachability analysis at the expense of doing more work during trace generation.

Suppose that during reachability analysis we simply annotate (p', γ', q) with a list of rules that has led to the addition of tuples in R'. Then, during reconstruction, we could still identify a set of predecessors for any configuration c which involves (p', γ', q). We need to pick one of them to continue the reconstruction. However, we are no longer guaranteed to pick a configuration which has been added to the automaton *before* c which destroys the guarantee that the reconstruction will terminate.

	no traces		complete		incomplete	
	time	mem	time	mem	time	mem
Regression suite	45.9 s	568 t	65.3 s	931 t	46.4 s	765 t
Four drivers	78.2 s	2673 t	305.4 s	6003 t	88.5 s	3216 t
Iscsiprt driver	119.2 s	4192 t	425.6 s	10593 t	142.4 s	5283 t
Serial driver	44.9 s	813 t	91.2 s	1998 t	50.7 s	924 t

Figure 5.4: Time and memory consumption for trace generation methods.

The solution Moped uses is to 'timestamp' the added transitions. Thus, the annotation becomes a list of items (r_i, c_i, R_i) where each R_i contains just the tuples which were first added by r_i, and c_i is the value of a global counter which is increased each time a list item is generated anywhere in the automaton. The reconstruction procedure can then search the list and make sure that it picks a predecessor configuration which was in the automaton before the current configuration.

The incomplete method creates an annotation with smaller R_is, but the list potentially contains many more items than in the complete method. Experiments, however, show that the incomplete method works faster and consumes less memory. The experiments were carried out on the Boolean Programs from Section 4.2.3. These were chosen for the experiments because trace generation was important for them (since the traces would be used to refine abstractions), because the traces that were generated could be quite long, and because the running times were dominated by the reachability and trace generation efforts. We compared the setting where no traces (and no annotations) were generated with the two methods for generating traces. Both time and memory in Figure 5.4 show the sums for each group of examples; memory consumption is measured in thousands of peak live BDD nodes.

The figures in the 'no traces' column include only resources needed for reachability, the others include those for reachability, generation of the annotation, and trace generation. Slightly surprisingly, the time for reachability and generating the annotation in the incomplete method was marginally less than the time for reachability when no traces were generated. This is because the relations needed in the annotations are computed during reachability anyway, and when not generating traces these are thrown away to save memory. Apparently this costs slightly more time than keeping them around and allocating some extra memory for the rest of the annotation.

5.4 The cost of generating shortest traces

In Section 3.1.7 we showed how one can compute the length of a shortest path from a set C to any configuration reachable from C by imposing a discipline on the way automaton transitions are chosen for processing. The discipline requires to maintain a priority queue for the transitions which are waiting to be processed, and inserting a transition into this queue is asymptotically more expensive than in the normal case where we use a stack.

Combining this technique with the one for reconstructing paths from Sections 3.1.6 and 5.3 lets us generate the shortest path from C to any configuration. It turns out that shortest paths do take noticeably longer to compute than arbitrary paths, but for other reasons than the cost of maintaining a priority queue. These reasons, which can be characterized as edge splitting and breadth-first search, are discussed below.

Edge splitting In Sections 3.1.6 and 3.1.7 we showed that the only changes required in order to make the *post** algorithm compute shortest paths are to compute a labelling for the transitions and to use a priority queue. Transferring these changes to the symbolic version of *post** (Algorithm 6 in Section 3.3.3) is not completely straightforward: In Algorithm 6 we maintain two sets $rel(q, \gamma, q')$ and $trans(q, \gamma, q')$ with each triple q, γ, q' to represent the transitions that have already been processed and those which still need to be. However, the tuples in $rel(q, \gamma, q')$ and $trans(q, \gamma, q')$ may be reached by shortest paths of different lengths and may therefore carry different labels. Therefore we need to partition the sets $rel(q, y, q')$ and $trans(q, y, q')$ so that each element set of the partition has the same label, and we need to process the element sets individually.

Breadth-first search The priority queue sorts transitions according to the length of a shortest path that has led to their creation. This basically means that we explore the reachable state space breadth-first. In the experiments with Boolean Programs, this mode of search usually forced the checker to explore a larger part of the state space before it found an error.

	default mode			shortest traces		
	time	tr. sets	length	time	tr. sets	length
Regression suite	46.4 s	36732	3786	58.4 s	63717	3549
Four drivers	88.5 s	169315	16559	126.9 s	283100	11886
Iscsiprt driver	142.4 s	79591	13111	243.4 s	152959	8377
Serial driver	50.7 s	9488	3371	60.2 s	39877	2620

Figure 5.5: Comparison of default vs shortest trace generation.

Experiments Moped's default mode for reachability analysis is to enqueue transitions using a stack, which (roughly speaking) means a depth-first exploration of the state space, and to report the first trace found in this way. We compared this method with the one for generating shortest traces in two respects: the time required by both methods, and the length of the generated trace. The examples used for comparison were again the four groups of Boolean Programs from Section 4.2.3. The table in Figure 5.5 lists the times for reachability and trace generation, the number of transition sets which were processed, and the length of the resulting trace.

On average, the shortest trace generation takes 50 percent more time than the default method. The traces generated by the default method are, on average, 40 percent longer than the shortest traces. Both numbers seem to be acceptable trade-offs in practice. The time spent for managing operations on the priority queue is marginal (less than a second in each group of examples), so the increase in time is mostly due to the increased number of transition sets which the shortest trace method must process (80 percent more).

To determine the source of the increase, we added edge splitting to the default method and ran the examples again. This resulted in a 4 percent increase in running time (for 17 per cent more transition sets). Therefore, our conclusion is that the increase is largely due to breadth-first search. We suspect that depth-first search is favourable for these examples because they are control-flow intensive, and only few of the statements influence the property that is being checked (e.g. the code may contain 'switch' statements with multiple cases, but none of the actions in these cases influences the property). This would increase the chance for a depth-first search to succeed, whereas a breadth-first search has to go through many 'useless' branches of the program.

	default mode		full reachability	
	pre*	*post**	*pre**	*post**
Regression suite	58.5 s	46.4 s	51.0 s	48.4 s
Four drivers	128.8 s	88.5 s	139.7 s	167.0 s
Iscsiprt driver	1864.0 s	142.4 s	2239.3 s	4663.6 s
Serial driver	127.0 s	50.7 s	161.4 s	460.9 s

Figure 5.6: Comparison of backward (*pre**) and forward (*post**) methods on Boolean Programs.

5.5 Backward versus forward computation

In Chapter 3 we have given solutions for both reachability and LTL model-checking problems based on backward and forward search. The backward solutions have a better complexity estimation, but the backward reachable state space may be larger than the state space reachable through forward search, forcing the backward methods to visit more states. We ran both the backward and the forward methods on the examples to see which aspect matters more in practice.

Reachability For the Boolean Programs, the forward method (using *post**) was better than the backward (*pre**) method on all four groups of examples by factors of 1.5, 13, and 2.5 on the various drivers. The numbers are listed in the left half of the table in Figure 5.6. These were recorded using Moped's default mode for reachability analysis which is to stop the search as soon as the result is known (i.e. when an error state has been reached from an initial state, or vice versa, or otherwise when reachability analysis has been completed without finding an error).

The large majority of the big examples (programs derived from drivers) were 'yes' instances, in which the error label was reachable so that the search could be aborted when the error was found. We ran another series of experiments where the reachability analysis was completed regardless of the result. The running times for these experiments are listed under "full reachability" in Figure 5.6. In these examples, *pre** was faster by factors of up to 2.5. Curiously enough, however, *pre** happened to be slower on the 'no' instances where the full reachability analysis was actually necessary.

K	N	M	*post**	*pre**
non-randomized				
2	4	8	10.2 s	3493.0 s
3	3	6	13.7 s	1314.1 s
randomized				
3	3	6	33.4 s	2811.8 s

Figure 5.7: Comparison of *pre** and *post** methods on Quicksort.

LTL model-checking For Quicksort (Section 4.2.1), the times for several sets of parameters using the forward method are listed in Figure 4.8. The backward method failed to terminate within an hour for most of the parameter sets; those where it did terminate are listed in Figure 5.7. This example is an extreme illustration of the point that the backward reachable state space may be much larger than the forward reachable space; for instance, the variable `left` is always less than or equal to `eq`, which is less than or equal to `lo` and so on. The backwards search does not 'know' this, and therefore does a lot of work in vain.

In Heapify (Section 4.2.1), however, the backward method was consistently faster than the forward method (see the table in Figure 4.9). This example has fewer restrictions to the forward reachable state space than Quicksort, which explains why *pre** can exploit its asymptotic advantage much better.

For the Unify example, times and memory consumption for the forward method are mentioned in Section 4.2.1. The backward method ran out of memory (1.5 GB) in all the tests.

Conclusions As a conclusion, the forward methods seem to be preferable, and they are used by default in Moped. The backward methods were always slower with one exception (Heapify) and sometimes failed to terminate within an hour when the forward method took only a couple of seconds. Even in the one case where the forward method was better, the forward method still terminated, being five times slower in the worst example.

In the Boolean Program examples, the backward (*pre**) method is slower to find the first error, but faster to compute the complete (backward) reachable state space. This suggests that the backward method computes too

	lazy	eager
Regression suite	46.4 s	47.3 s
Four drivers	88.5 s	91.2 s
Iscsiprt driver	142.4 s	152.1 s
Serial driver	50.7 s	54.9 s

Figure 5.8: Running times for eager and lazy evaluation.

many transitions which do not lead towards the initial states. In fact, by processing the rules with an empty right-hand side first, the pre^* algorithm effectively starts its computation at the return statements of all procedures simultaneously and works its way forward from there. It may well be that one can achieve better performance by tailoring the pre^* algorithm to Boolean Programs, and start at return statements only if a call to their procedure has been shown to be backwards reachable from the error label. This could be the subject of further research.

5.6 Eager versus lazy evaluation

In Section 3.1.5 we considered the case where a pushdown system is derived from a Boolean Program and analyzed the actions of the $post^*$ algorithm in terms of the program. Comparing this analysis with that of similar algorithms [1, 3, 6] revealed that interactions between procedures can be evaluated in two ways: lazy and eager.

Algorithm 2 from Section 3.1.4 uses lazy evaluation. To understand the impact of the mode of evaluation on performance, let us consider an alternative eager variant of $post^*$ (Algorithm 7 in Figure 5.6). The essential difference to Algorithm 2 is the addition of lines 21 through 23; here the 'procedure call' rules are combined with the fact that the result of a procedure call has been evaluated (represented by (p, ε, q)) to form new, 'artificial' rules. These new rules have the effect that was achieved by lines 17 and 18 in Algorithm 2.

The Boolean Program examples were used to compare the lazy and the eager version. The resulting running times in seconds are listed in Figure 5.8. The differences are minor, but eager evaluation is consistently slower at an average of 5 percent. The slower performance of eager evaluation is expected because it does too much work: Whenever the effect of a procedure is computed, the effect is plugged into all possible call sites. Some of these call sites are never visited, so the work done for them is in vain.

The concepts of lazy and eager evaluation can also be applied to backward reachability. The algorithm presented for *pre** (Algorithm 1 in Section 3.1.3) uses eager evaluation. The author believes that that eager evaluation works better for *pre** – see the justification given at the beginning of Section 3.1.3 – but no experiments have been carried out to confirm this suspicion. There seems to be little gain to be had from such a comparison, though, as backward reachability is often slower than its forward counterpart (see Section 5.5), and the expected small difference between eager and lazy evaluation is unlikely to change this.

5.7 Variable optimizations

The model-checking of Boolean Programs could benefit from techniques to reduce the size of the BDDs used to represent the programs. Ball and Rajamani [3] proposed such techniques based on ideas known from compiler optimization.

Mod/ref analysis Let x be a global variable and p be a procedure. Suppose that neither p nor any of the procedures called by p (either directly or transitively) modifies or references x. Then it is not necessary to record the value of x inside p, provided that x is saved on the stack during any call to p. This way, the BDDs representing statements of p need not mention the BDD variables for x.

Moped implements this technique with the following addition: If p contains a *constrain* statement (see the syntax description in the Appendix), then the structure of its associated expression is examined. If the expression contains a conjunctive clause of the form $x = {}'x$, this clause merely preserves the value of x and is not counted as a reference to x.

Live range analysis A variable x of a Boolean Program is said to be *dead* at program point n if, on all paths starting at n, the value of x is never used before x is modified. If x is dead at program point n, then the program can discard the value of x when reaching n. In BDD terms, this means that the transition relations of statements at n place no condition

Algorithm 7
Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$;
a $\mathcal{P}$-Automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ without transitions into P
Output: the automaton $\mathcal{A}_{post^*}$

$\mathit{trans} := (\rightarrow_0) \cap (P \times \Gamma \times Q)$;
$\mathit{rel} := (\rightarrow_0) \setminus \mathit{trans}$; $Q' := Q$; $F' := F$; $\Delta' := \emptyset$;
for all $\langle p, \gamma \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_2 \rangle \in \Delta$ **do**
  $Q' := Q' \cup \{q_{p_1,\gamma_1}\}$;
for all $q \in Q'$ **do** $\mathit{eps}(q) := \emptyset$;
while $\mathit{trans} \neq \emptyset$ **do**
  pop $t = (p, \gamma, q)$ from trans;
  if $t \notin \mathit{rel}$ **then**
    $\mathit{rel} := \mathit{rel} \cup \{t\}$;
    if $\gamma \neq \varepsilon$ **then**
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \varepsilon, q)\}$
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \rangle \in (\Delta \cup \Delta')$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \gamma_1, q')\}$;
      for all $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \gamma_2 \rangle \in \Delta$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p', \gamma_1, q_{p',\gamma_1})\}$;
        $\mathit{rel} := \mathit{rel} \cup \{(q_{p',\gamma_1}, \gamma_2, q)\}$;
    else
      for all $(q, \gamma', q') \in \mathit{rel}$ **do**
        $\mathit{trans} := \mathit{trans} \cup \{(p, \gamma', q')\}$;
      if $q = q_{p_1,\gamma_1} \in Q' \setminus Q$ **then**
        for all $\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_3 \rangle \in \Delta$ **do**
          $\Delta' := \Delta' \cup \{\langle p_2, \gamma_2 \rangle \hookrightarrow \langle p, \gamma_3 \rangle\}$;
      if $q \in F'$ **then** $F' := F' \cup \{p'\}$;
return $(Q', \Gamma, \mathit{rel}, P, F')$

Figure 5.9: A 'eager' variant of $post^*$.

	none	mod/ref	live range	both
Regression suite	45.9 s	44.1 s	45.0 s	43.5 s
Four drivers	78.2 s	75.5 s	67.4 s	66.7 s
Iscsiprt driver	119.2 s	114.9 s	105.4 s	101.7 s
Serial driver	44.9 s	48.7 s	50.5 s	49.5 s

Figure 5.10: Comparison of variable optimizations.

on the pre/post values of x so that their BDDs become smaller. Moped implements this technique for local variables; this requires an inspection of just the statements of the respective procedure.

Experimental results We tested the impact of the optimizations on the Boolean Program examples both individually and in combination. The running times are listed in Figure 5.10; these do not include time for trace generation (see below for the reason). Both optimizations managed to reduce the time for the model-checker, although not dramatically (up to 15 percent combined). In the serial driver examples their use was counterproductive because for these large programs the time taken to compute the information needed for the optimizations could not be made up during the relatively short reachability analysis.

Unfortunately, the optimizations conflict with trace generation: When the pushdown model discards dead variables or saves global variables on the stack (and discards them until they are retrieved), the mechanics of the model are no longer a completely faithful representation of the semantics of the underlying Boolean Program. As a result, an error trace may for instance change the value of a variable from 0 to 1 although the program does not warrant such a change. Although such incidents are 'harmless' in the sense that they do not lead to false error reports, they may confuse users who are reading the trace. They also confuse Newton, the SLAM tool which checks whether traces are spurious (see Section 4.2.3).

The incompatibility between the optimizations and trace generation is not a fundamental problem – if the model-checker keeps track of which variables are discarded at which points in the model, then the trace can be corrected. The checker can go through the trace step by step starting at its initial configuration; if a variable changes its value in the trace when it is in a discarded state, then the change is 'illegal' and the variable must retain its value. However, at the time of writing such a correction is not implemented in Moped; hence, it is not recommended to use the optimizations when traces are required.

5.8 Variable ordering

It is well known that the performance of BDD-based algorithms is very sensitive to variable ordering. This section describes the factors which were essential in obtaining good variable order for the examples. We discuss the issues of arrays and integer variables in the 'academic' examples (Quicksort, Heapify, Unify), and a dependency-based ordering mechanism for the Boolean Programs.

Arrays The Quicksort, Heapify, and Unify examples use scalar variables (booleans or integers, the latter represented by multiple BDD variables), and arrays. Typically, scalar variables are used as indices in the array, or used for comparison with array elements. We shall argue that it is better for arrays to occur *after* the scalar variables in the variable ordering.

Consider, for instance, the expression $a[i] = p$ where a is an array with M elements and i and p are scalars. Moreover, let us assume that p and the elements of a have K bits such that they have a range between 1 and $D := 2^K$. Figure 5.11 shows a schematic view of two BDDs for the expression; black nodes indicate the 1-leaf, and edges to the 0-leaf are not shown. The first BDD assumes the ordering i, p, a (the scalar variables before the array), the second a, i, p (vice versa).

Let us compare the width of the BDDs (i.e. the maximum number of nodes at the same level) as a measure of their sizes. For both cases, the width is indicated by the number of grey nodes. Under the first ordering, the width is $M \cdot D$ because the BDD has to branch into a different case for each possible value of i and p. In the lower part of the first BDD, the paths to the 1-leaf ensure that the array element selected by i indeed has the same value as p which requires K BDD nodes in each of the $M \cdot D$ cases. In the second ordering, the BDD has to split into D^M different cases, one for every possible array content – each case allows a different set of combinations between i and p to reach the 1-leaf.

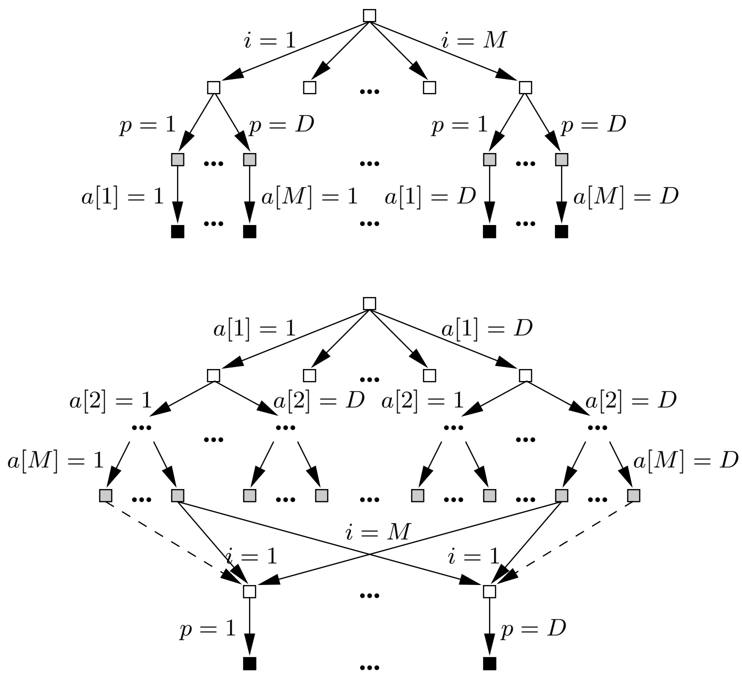

Figure 5.11: BDDs for $a[i] = p$ with orderings i, p, a (above) and a, i, p (below).

Under the first ordering, the number of nodes required for the expression is linear in M, under the second ordering it is exponential. Similar results could be obtained, e.g., for expressions of the form $a[i] = a[j]$. In the experiments, using the first type of ordering was essential to avoid running out of memory for most examples.

Integers When integer variables are represented by multiple BDD variables, it is usually beneficial to order the BDD variables from most to least significant bit; this minimizes the BDDs used to represent arithmetic comparisons. If several integer variables are present in a program, we have two choices for the ordering: blockwise, i.e. first all BDD variables for the first integer, then all for the second integer etc, or interleaved, i.e. first the most significant bit for all integers, then the second most etc. until the least significant bit (given the result of the previous paragraph, we first interleave all scalar variables and then the arrays).

In the experiments, blockwise or interleaved ordering made no apprecia-

			blockwise	interleaved
`K`	`N`	`M`	Quicksort	
2	4	12	66.8 s	493.2 s
3	3	6	20.1 s	13.7 s
4	4	6	654.8 s	64.4 s
4	4	7	*	449.4 s
`K`		`M`	Heapify	
2		11	14.4 s	11.8 s
3		9	646.3 s	36.0 s
3		11	*	1938.1 s

	blockwise	interleaved
Unify w/o occurs		
depth 2	61.5 s	112.7 s
depth 3	249.5 s	579.2 s
Unify with occurs		
small (2/1/1)	181.3 s	556.3 s
large (3/2/3)	6126.5 s	*

Figure 5.12: Blockwise and interleaved ordering of arrays.

ble difference for the scalar variables. For the arrays, both ordering modes had advantages and disadvantages, depending on the nature of the examples. Figure 5.12 lists some results; asterisks indicate that the experiment ran out of memory. Blockwise ordering was better for the Unify examples; in the Quicksort and Heapify examples interleaved ordering outperforms blockwise ordering as `K`, the number of bits used for representing integers, increases. This can be explained with the observation that these two examples compute permutations (i.e., the BDDs in the automata express a relation between the original array contents and the resulting contents, the latter being a permutation of the former), and that interleaved ordering can express permutations with smaller BDDs.

Boolean Programs The requirements for the Boolean Program examples are different from the 'academic' examples. The issues discussed above do not matter, since Boolean Program have only scalar boolean variables, but no arrays or integers. However, the programs may contain hundreds of such variables, so finding a good ordering for them is still important. To this aim, Moped employs a heuristic borrowed from Bebop [3].

The heuristic determines a variable ordering based on dependency analysis. We say that program variable a_1 *depends* on program variable a_2 if and only if the program contains at least one statement where a_1 is computed as the value of an expression e which contains a_2. In Boolean Programs, this can happen in three cases:

- there is an assignment $a_1 := e$;
- a_1 is a formal parameter of a function f, and f is called with expression e as an argument in the place for a_1;
- a_1 takes a return value from a function call, and the callee contains a return statement which specifies e in the place for a_1.

The ordering tries to incorporate two ideas:

(i) if two or more variables have a dependency relation with each other, they should be placed close to each other in the ordering.

(ii) each variable should be placed after those variables upon which it depends.

Concretely, the heuristic comprises the following steps:

1. Assign a BDD variable to each program variable (see Section 5.9).
2. Build the *dependency graph*. The dependency graph is a directed graph with one node for each BDD variable; if program variable a_1 depends on a_2, then there is an edge from the BDD variable for a_1 to the BDD variable for a_2.
3. If there is a cyclic dependency, remove any one of the edges which form part of a cycle. Repeat this until all cycles have been eliminated.
4. Compute the weight of each node as the number of its (transitive) children. Sort the nodes in descending order of weight. Mark all nodes as unvisited.
5. As long as there are unvisited nodes left, pick an unvisited node v with maximal weight. Start a depth-first traversal of the graph at v, limiting the traversal to previously unvisited nodes. During the traversal, add the associated BDD variables to the ordering in postorder fashion. Mark all visited nodes as such. Repeat this step until all nodes have been visited (and all variables added to the ordering).

	heuristic	input order
Regression suite	46.4 s	47.6 s
Four drivers	88.5 s	153.5 s
Iscsiprt driver	142.4 s	270.5 s
Serial driver	50.7 s	$*$

Figure 5.13: Effect of variable ordering heuristic.

Idea (ii) is reflected in sorting nodes by weight; if a_1 depends on a_2, then (barring removed edges) the weight of a_1 will be larger than that of a_2, and adding variables in postorder places a_2 ahead of a_1. Idea (i) is reflected by using depth-first traversal (instead of simply sorting nodes/BDD variables in ascending order of weight which would also satisfy idea (ii)).

Figure 5.13 details how the heuristic affected computation times; we compare it against 'input order', i.e. the case where the variable orderings simply follow the order of declarations. As one could expect, the ordering hardly matters on the small regression suite examples. Most drivers proved surprisingly independent of the variable ordering as, on average, the heuristic only reduced their running time by half. The heuristic did make a huge difference for the serial drivers, however, as the input-based ordering ran out of memory on two of the four examples in that group. Closer analysis revealed that this happens because these examples contain functions with fairly large numbers of arguments and return values (see Figure 4.14). These may result in parallel assignments between two large groups of variables, and the BDDs for these are only manageable with good orderings.

5.9 Variable mapping strategies

Each procedure in a Boolean Program has its own set of local variables. Similarly, the input language for symbolic pushdown systems allows sets of stack symbols to be associated with different sets of local variables. This section concerns the question of how the local variable spaces should be represented symbolically. Moped employs three different strategies:

- "avaricious": As we have seen, the complexity of the associated model-checking problems depends only on the size of the *largest* local variable space. This method determines the number of bits necessary to represent the largest local variable set and uses only that many BDD variables (i.e. no more BDD variables than absolutely necessary).

- "generous": Allocate a fresh set of BDD variables for each set of local variables.
- "by name": This mode is a compromise between the two others: if two local variables in different sets have the same name and type, they are represented by the same BDD variables. The motivation for this mode is that variables of the same name and type often perform the same tasks.

Notice that none of these modes changes the asymptotic complexity of any of the algorithms (at least theoretically, see below) because the number of different BDD variables which occur in any individual BDD is not changed, we just change the way different sets are represented. The advantage of using fewer BDD variables is that if the same BDD variables are used to represent multiple program variables, then less memory may be needed because of node sharing in the BDD library. Another important point is that the performance of some BDD libraries deteriorates when they have to manage large numbers of variables even if BDD sizes do not increase. These two points are demonstrated to an extreme extent on a family of examples in Section 4.2.2. On the other hand, an advantage of using more BDD variables is that it becomes easier to determine a good variable ordering. In terms of the heuristic presented in Section 5.8, a generous allocation strategy leads to a sparse dependency graph, an avaricious strategy to a dense graph with possibly many cycles.

We compare the strategies by their performance on the Boolean Program examples. The results on time and memory consumption are detailed in Figure 5.14 (the figures for memory list the average in each group; the proportions between average and maximum were always roughly the same between the three methods).

The numbers demonstrate the pros and cons of the methods mentioned above; memory consumption is generally lower the less variables are used, and fewer variables also speed up the BDD operations. However, the avaricious method ran out of memory in two of the four serial driver examples (the largest examples in the tests) because no sufficiently good variable ordering could be determined anymore (compare also the failures on the same

	avaricious		by name		generous	
Regression suite	44.8 s	4.3 M	46.4 s	4.3 M	46.8 s	4.3 M
Four drivers	87.0 s	7.0 M	88.5 s	7.6 M	91.0 s	9.4 M
Iscsiprt driver	132.5 s	12.4 M	142.4 s	14.9 M	162.6 s	23.5 M
Serial driver	$*$	$*$	50.7 s	23.2 M	64.8 s	41.3 M

Figure 5.14: Effect of variable mapping strategies.

examples in Section 5.8).

Based on these experiments, the 'by name' method appears to be the most favourable, and it has been chosen as the default in Moped. The avaricious method fails on the largest group of examples, and the generous method takes much longer on the examples in Section 4.2.2. Apart from these, however, the differences between the three methods are not dramatic.

Chapter 6

Regular Valuations

In this chapter we revisit the model-checking problem for LTL and pushdown systems. In Chapter 3 we stated that the problem is undecidable for arbitrary valuations, i.e., the functions that map the atomic propositions of a formula to the sets of pushdown configurations which satisfy them. However, it was shown that the problem remains decidable for *simple* valuations, which are completely determined by the control location and the topmost stack symbol. Here, we extend the solution to *regular* valuations which depend on regular predicates over the complete stack content.

This extension is motivated by several applications, for instance interprocedural data-flow analysis. Here, regular valuations can be used to compute data-flow information which dynamically depends on the history of procedure calls. A second application area are systems with checkpoints. In these systems computation is suspended at certain points to allow for a property of the stack content to be checked; resumption of the computation depends on the result of this inspection. This allows, for instance, to model programs with security checks in which certain actions are allowed only if the procedures on the stack have appropriate permissions. A third application allows to extend the model-checking algorithms from Chapter 3 to the stronger logic CTL*.

The solutions for regular valuations are based on a reduction to the case of simple valuations, which allows to re-use most of the theory from Chapter 3. However, special attention is payed to ensuring the efficiency of the reduction. Using regular valuation increases the complexity of the model-checking algorithms. The factor by the complexity increases depends on the size of the finite automata which express the regular predicates. A straightforward reduction and analysis would yield a cubic blowup for problems (III) and (IV) and even a quadric blowup for problem (V), see Theorems 3.9 and 3.10. However, by modifying the algorithms slightly and by exploiting special properties of the constructions the blowup can be reduced to a *linear* factor in terms of both time and space.

In principle, two different techniques can be used for this purpose; one works by extending the control locations, the other by extending the stack alphabet. Both techniques lead to the same asymptotic (linear) blowup; however, the technique of extending the stack alphabet works for regular valuations in general whereas that of extending the control locations works only for a restricted subclass.

Following the presentation of the techniques, the potential applications are examined in more detail in Sections 6.4, 6.5, and 6.6. Finally, Section 6.7 takes another look at the complexity of the methods. The algorithms are polynomial in the size of a deterministic finite automaton which encodes the regular valuations. Since this automaton is the product of the automata which encode the individual atomic propositions, its worst-case size can be considered exponential in the size of the input. However, it can be shown that there exist lower bounds for these problems which imply that no polynomial solution exists.

6.1 Regular Valuations

In Section 3.2 we introduced the concept of valuations to provide the formal meaning of an LTL formula in the context of a transition system. Given a formula φ and a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$, a valuation function $\nu\colon At(\varphi) \to 2^{Conf(\mathcal{P})}$ assigns to each atomic proposition the configurations which satisfy it. We define the class of *regular valuations* as follows:

Definition 6.1 *Let $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ be a pushdown system and φ an LTL formula. A valuation $\nu\colon At(\varphi) \to 2^{P\times\Gamma^*}$ is* regular *if $\nu(A)$ is a regular set for every $A \in At(\varphi)$ and does not contain any configuration with empty stack.*

The requirement that configurations with empty stack cannot satisfy any atomic proposition simplifies the following presentation. Since we can always extend a pushdown system with an artificial 'bottom-of-stack' symbol, the requirement is not a real restriction. Since regular sets can be infinite, we need to represent regular valuations by finite means. We fix an adequate representation for our purposes.

Definition 6.2 *A finite automaton* $\mathcal{A} = (Q, \Gamma, \rightarrow, q_0, F)$ *is* deterministic *if for each pair* $q \in Q$, $\gamma \in \Gamma$ *there is at most one* $q' \in Q$ *such that* $q \xrightarrow{\gamma} q'$. *We call* $\mathcal{A}$ backward deterministic *if for each pair* $q \in Q$, $\gamma \in \Gamma$ *there is at most one* $q' \in Q$ *such that* $q' \xrightarrow{\gamma} q$.

Notation Let ν be a regular valuation. For every atomic proposition A and control location p, we denote by $\mathcal{M}_A^p$ a deterministic finite-state automaton over the alphabet Γ with a total transition function and satisfying

$$\nu(A) = \{\, \langle p, w \rangle \mid p \in P, w^R \in L(\mathcal{M}_A^p) \,\}$$

where w^R denotes the reverse of w.

Hence, A is true at $\langle p, w \rangle$ if and only if the automaton $\mathcal{M}_A^p$ enters a final state after reading the stack contents bottom-up. Since $\nu(A)$ does not contain any configuration with empty stack, the initial state of $\mathcal{M}_A^p$ is not accepting.

For simple valuations, the model-checking problems (III) to (V) proposed in Section 2.5 reduce to constructing a Büchi pushdown system $\mathcal{BP}$ as the product of $\mathcal{P}$ and a Büchi automaton for the negation of the formula, and to solving various reachability problems on $\mathcal{BP}$ (see Section 3.2). Our objectives therefore are to provide analogous methods for the case of regular valuations. We show that one can actually reduce the problem to model-checking simple valuations.

In the next two sections we fix a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, \langle p_0, w_0 \rangle)$, an LTL formula φ, and a regular valuation ν. The Büchi automaton which corresponds to $\neg\varphi$ is denoted by $\mathcal{B} = (R, 2^{At(\varphi)}, \eta, r_0, G)$. Let $At(\varphi) = \{A_1, \ldots, A_n\}$, and let $\mathcal{M}_{A_i}^p = (Q_i^p, \Gamma, \rightarrow_{i,p}, s_i^p, F_i^p)$ be the deterministic finite automaton associated to (A_i, p) for all $p \in P$ and $1 \leq i \leq n$.

We present two techniques for model-checking with regular valuations. Both of them reduce the problem to model-checking with simple valuations and in both cases, the idea is to encode the $\mathcal{M}_{A_i}^p$ automata into the structure of $\mathcal{P}$ and simulate them on the fly during the computation of $\mathcal{P}$. In practice, some of the $\mathcal{M}_{A_i}^p$ automata can be identical. Of course, it is unnecessary to simulate the execution of the *same* automaton within $\mathcal{P}$ twice, and the constructions should reflect this. For simplicity, we assume that whenever $i \neq j$ or $p \neq q$, then the $\mathcal{M}_{A_i}^p$ and $\mathcal{M}_{A_j}^q$ automata are either identical or have disjoint sets of states. So, let $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ be the *set* of all $\mathcal{M}_{A_i}^p$ automata where $1 \leq i \leq n$ and $p \in P$ (hence, if some of the $\mathcal{M}_{A_i}^p$ automata are identical, then $m < n \cdot |P|$), and let Q_j be the states of $\mathcal{M}_j$ for each $1 \leq j \leq m$. The Cartesian product $Q_1 \times \cdots \times Q_m$ is denoted by *States*. For given $\mathbf{r} \in \mathit{States}$, $p \in P$, and $1 \leq i \leq n$, we denote by $\mathbf{r}_i^p$ the element of Q_i^p which appears in $\mathbf{r}$ (observe that we can have $\mathbf{r}_i^p = \mathbf{r}_j^q$ even if $i \neq j$ or $p \neq q$). The vector of initial states (i.e., the only element of *States* where each component is the initial state of some $\mathcal{M}_{A_i}^p$) is denoted by $\mathbf{s}$. Furthermore, we write $\mathbf{r} \xrightarrow{\gamma}_{\mathcal{M}} \mathbf{t}$ if $\mathbf{r}_i^p \xrightarrow{\gamma}_{i,p} \mathbf{t}_i^p$ for all $1 \leq i \leq n$ and $p \in P$.

Remark 6.1 (On the complexity measures) *The size of an instance of the model-checking problem for pushdown systems and LTL with regular valuations is given by* $|\mathcal{P}| + |\varphi| + |\nu_\varphi|$, *where* $|\nu_\varphi|$ *is the total size of all employed automata. However, in practice we usually work with small formulas and a small number of simple automata (see Sections 6.4 and 6.5); therefore, we measure the complexity of our algorithms in* $|\mathcal{B}|$ *and* $|\mathit{States}|$ *rather than in* $|\varphi|$ *and* $|\nu_\varphi|$. *In general, however,* $\mathcal{B}$ *and States can be* exponentially *larger than* φ *and* ν_φ. *This allows for a detailed complexity analysis whose results better match the reality because* $|\mathcal{B}|$ *and* $|\mathit{States}|$ *often stay small. Section 6.7 provides some lower bounds which show that all algorithms developed in this paper are also essentially optimal from the point of view of worst-case analysis.*

6.2 Technique 1 – extending the finite control

The idea behind this technique is to evaluate the truth value of the atomic propositions $A_1, \ldots, A_n$ on the fly by storing the product of $\mathcal{M}_{A_i}^p$ automata in the finite control of $\mathcal{P}$ and updating the vector of states after each transition according to the change of stack contents. We will use the assumptions that the $\mathcal{M}_{A_i}^p$ automata are deterministic, have total transition functions, and read the stack bottom-up. However, we also need one additional assumption to make the construction work:

Each automaton $\mathcal{M}_{A_i}^p$ is also backward deterministic.

This assumption is truly restrictive – there are quite simple regular languages which cannot be recognized by finite-state automata which are both deterministic and backward deterministic, e.g. the language $\{\, a^i \mid i > 0 \,\}$.

We define a pushdown system $\mathcal{P}' = (P', \Gamma, \Delta', c_0')$ where $P' = P \times \mathit{States}$, $c_0' = \langle (p_0, \mathbf{r}), w_0 \rangle$, and $\mathbf{r}$ is the unique state determined by $\mathbf{s} \xrightarrow{w_0^R}{}^{*}_{\mathcal{M}} \mathbf{r}$. The transition rules Δ' contain a rule $\langle (p, \mathbf{r}), \gamma \rangle \hookrightarrow_{\mathcal{P}'} \langle (p', \mathbf{u}), w \rangle$ if and only if the following conditions hold:

- $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$,
- there is $\mathbf{t} \in \mathit{States}$ such that $\mathbf{t} \xrightarrow{\gamma}_{\mathcal{M}} \mathbf{r}$ and $\mathbf{t} \xrightarrow{w^R}{}^{*}_{\mathcal{M}} \mathbf{u}$.

Observe that due to the backward determinism of $\mathcal{M}^p_{A_i}$ there is at most one $\mathbf{t}$ with the above stated properties; and thanks to determinism and the totality of $\rightarrow_{i,p}$ we further obtain that for given $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$ and $\mathbf{r} \in \mathit{States}$ there is exactly one $\mathbf{u} \in \mathit{States}$ such that $\langle (p, \mathbf{r}), \gamma \rangle \hookrightarrow_{\mathcal{P}'} \langle (p', \mathbf{u}), w \rangle$. From this it follows that $|\Delta'| = |\Delta| \cdot |\mathit{States}|$.

We shall call a configuration $\langle (p, \mathbf{r}), w \rangle$ of $\mathcal{P}'$ *consistent* if $\mathbf{s} \xrightarrow{w^R}{}^{*}_{\mathcal{M}} \mathbf{r}$. In other words, $\langle (p, \mathbf{r}), w \rangle$ is consistent if and only if $\mathbf{r}$ 'reflects' the stack contents w. Let $\langle p, w \rangle$ be a configuration of $\mathcal{P}$ and $\langle (p, \mathbf{r}), w \rangle$ be the (unique) associated consistent configuration of $\mathcal{P}'$. Now we can readily confirm that

(A) if $\langle p, w \rangle \Rightarrow_{\mathcal{P}} \langle p', w' \rangle$, then $\langle (p, \mathbf{r}), w \rangle \Rightarrow_{\mathcal{P}'} \langle (p', \mathbf{u}), w' \rangle$ where $\langle (p', \mathbf{u}), w' \rangle$ is the unique consistent configuration associated with $\langle p', w' \rangle$;

(B) if $\langle (p, \mathbf{r}), w \rangle \Rightarrow_{\mathcal{P}'} \langle (p', \mathbf{u}), w' \rangle$, then $\langle (p', \mathbf{u}), w' \rangle$ is consistent and $\mathcal{T}_{\mathcal{P}}$ has a transition $\langle p, w \rangle \Rightarrow_{\mathcal{P}} \langle p', w' \rangle$.

As the initial configuration of $\mathcal{P}'$ is consistent, we see due to (B) that each reachable configuration of $\mathcal{P}'$ is consistent. Moreover, due to (A) and (B) we also have the following:

(C) let $\langle p, w \rangle$ be a configuration of $\mathcal{P}$ (not necessarily reachable) and let $\langle (p, \mathbf{r}), w \rangle$ be its associated consistent configuration of $\mathcal{P}'$. Then the parts of $\mathcal{T}_{\mathcal{P}}$ and $\mathcal{T}_{\mathcal{P}'}$ which are reachable from $\langle p, w \rangle$ and $\langle (p, \mathbf{r}), w \rangle$, respectively, are isomorphic.

We define the simple valuation ν' by

$$\nu'(A_i) = \{\, \langle (p, \mathbf{r}), w \rangle \mid (p, \mathbf{r}) \in P',\ \mathbf{r}^p_i \in F^p_i,\ w \in \Gamma^+ \,\}$$

for all $1 \leq i \leq n$. Due to (C) it is easy to see that for all $p \in P$ and $w \in \Gamma^*$ we have

$$\langle p, w \rangle \models^{\nu} \varphi \iff \langle (p, \mathbf{r}), w \rangle \models^{\nu'} \varphi \text{ such that } \mathbf{s} \xrightarrow{w^R}{}^{*}_{\mathcal{M}} \mathbf{r} \qquad (*)$$

We observed that $|P'| = |P| \cdot |States|$ and $|\Delta'| = |\Delta| \cdot |States|$. Applying Theorems 3.9 and 3.10 naïvely, we obtain that using Technique 1, the model-checking problems (III) and (IV) can be solved in cubic time and quadratic space (w.r.t. $|States|$), and that problem (V) takes quadric time and space. However, closer analysis reveals that we can do much better:

Theorem 6.1 *Let n_2 be the number of different pairs $(p', \gamma') \in P \times \Gamma$ such that a rule $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', \gamma'\gamma'' \rangle$ exists. Technique 1 (extending the finite control) gives us the following bounds on the model-checking problems with regular valuations:*

1. *Problems (III) and (IV) can be solved in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^2)$ space.*

2. *Problem (V) can be solved in either $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2) \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2) \cdot |States| \cdot |\mathcal{B}|^2)$ space, or $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + n_2) \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + n_2) \cdot |States| \cdot |\mathcal{B}|^2)$ space.*

In other words, all problems take only *linear* time and space in $|States|$. We skip the (somewhat lengthy) proof here; it is given in full detail in [20].

6.3 Technique 2 – extending the stack

An alternative approach to model-checking with regular valuations is to store the vectors of *States* in the stack of $\mathcal{P}$. This technique works without any additional limitations, i.e., we do *not* need the assumption of backward determinism of $\mathcal{M}^p_{A_i}$ automata.

Let $\mathcal{P}' = (P, \Gamma', \Delta', \langle p_0, w'_0 \rangle)$ be a pushdown system where $\Gamma' = \Gamma \times States$, and the set of transition rules Δ' is determined as follows:

- $\langle p, (\gamma, \mathbf{r}) \rangle \hookrightarrow_{\mathcal{P}'} \langle p', \varepsilon \rangle \iff \langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', \varepsilon \rangle$
- $\langle p, (\gamma, \mathbf{r}) \rangle \hookrightarrow_{\mathcal{P}'} \langle p', (\gamma', \mathbf{r}) \rangle \iff \langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', \gamma' \rangle$

- $\langle p, (\gamma, \mathbf{r})\rangle \hookrightarrow_{\mathcal{P}'} \langle p', (\gamma', \mathbf{u})(\gamma'', \mathbf{r})\rangle \iff \langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', \gamma'\gamma''\rangle \wedge \mathbf{r} \xrightarrow{\gamma''}_{\mathcal{M}} \mathbf{u}$

If $w_0 = \gamma_n \ldots \gamma_1$, the initial stack content is $w_0' = (\gamma_n, \mathbf{t}_n) \ldots (\gamma_1, \mathbf{t}_1)$ where $\mathbf{t}_i,\ 1 \leq i \leq n$, are the unique states defined by $\mathbf{s} \xrightarrow{\gamma_1 \cdots \gamma_{i-1}}{}^{*}_{\mathcal{M}} \mathbf{t}_i$.

Intuitively, the reason why we do not need the assumption of backward determinism in this approach is that the stack carries complete information about the computational history of the $\mathcal{M}^p_{A_i}$ automata.

A configuration $\langle p, (\gamma_k, \mathbf{r}_k) \ldots (\gamma_1, \mathbf{r}_1)\rangle$ is called *consistent* if and only if $\mathbf{r}_1 = \mathbf{s}$ and $\mathbf{r}_j \xrightarrow{\gamma_j}_{\mathcal{M}} \mathbf{r}_{j+1}$ for $1 \leq j < k$. The simple valuation ν' is given by

$$\nu'(A_i) = \{\, \langle p, (\gamma, \mathbf{r})w\rangle \mid p \in P,\ \gamma \in \Gamma,\ w \in \Gamma'^{*},\ \exists q_f \in F_i^p \colon \mathbf{r}_i^p \xrightarrow{\gamma}_{i,p} q_f \,\}$$

It is easy to see that $\langle p, \gamma_k \ldots \gamma_1\rangle \models^{\nu} \varphi \iff \langle p, (\gamma_k, \mathbf{r}_k) \ldots (\gamma_1, \mathbf{r}_1)\rangle \models^{\nu'} \varphi$ where $\langle p, (\gamma_k, \mathbf{r}_k) \ldots (\gamma_1, \mathbf{r}_1)\rangle$ is consistent.

Theorem 6.2 *Technique 2 (extending the stack) gives us the same bounds on the model-checking problems with regular valuations as Technique 1, i.e.:*

1. *Problems (III) and (IV) can be solved in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^2)$ space.*
2. *Problem (V) can be solved in either $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2) \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2) \cdot |States| \cdot |\mathcal{B}|^2)$ space, or $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + n_2) \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + n_2) \cdot |States| \cdot |\mathcal{B}|^2)$ space.*

Proof: Since $|\Delta'| = |\Delta| \cdot |States|$ (using the fact that each $\mathcal{M}^p_{A_i}$ is deterministic), we can compute a $\mathcal{P}$-automaton $\mathcal{A} = (P \cup \{s\}, \Gamma', \rightarrow, P, \{s\})$ of size $\mathcal{O}(|P| \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^2)$ in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^2)$ space such that $\mathcal{A}$ recognizes all configurations of $\mathcal{P}'$ which violate φ (see Theorem 3.9); then, to solve problem (III), we just look if $\mathcal{A}$ accepts $\langle p_0, w_0'\rangle$. The problem with $\mathcal{A}$ is that it can also accept inconsistent configurations. Fortunately, it is possible to perform a synchronization with the reversed $\mathcal{M}^p_{A_i}$ automata. We define

$$\mathcal{A}' = ((P \cup \{s\}) \times States \cup P, \Gamma, \rightarrow', P, \{s, \mathbf{s}\})$$

where $\rightarrow'$ is defined as follows:

- if $g \xrightarrow{(\gamma,\mathbf{r})} h$ and $\mathbf{r} \xrightarrow{\gamma}_{\mathcal{M}} \mathbf{t}$, then $(g, \mathbf{t}) \xrightarrow{\gamma}' (h, \mathbf{r})$;

- if $(p, \mathbf{r}) \xrightarrow{\gamma} (g, \mathbf{t})$, $p \in P$, then $p \xrightarrow{\gamma}' (g, \mathbf{t})$.

Notice that $\mathcal{A}'$ has the same asymptotic size as $\mathcal{A}$ since in every transition $\mathbf{t}$ is uniquely determined by $\mathbf{r}$ and γ. Now, we can easily prove (by induction on k) that for every configuration $\langle p, (\gamma_k, \mathbf{r}_k) \ldots (\gamma_1, \mathbf{r}_1) \rangle$ we have

$$p \xrightarrow{(\gamma_k,\mathbf{r}_k)\ldots(\gamma_1,\mathbf{r}_1)}{}^{*}\ q \text{ where } \mathbf{r}_j \xrightarrow{\gamma_j}_{\mathcal{M}} \mathbf{r}_{j+1} \text{ for all } 1 \leq j < k$$

if and only if

$$(p, \mathbf{r}) \xrightarrow{\gamma_k \cdots \gamma_1}{}'^{*}\ (q, \mathbf{r}_1) \text{ where } \mathbf{r}_k \xrightarrow{\gamma_k}_{\mathcal{M}} \mathbf{r}$$

From this we immediately obtain that $\mathcal{A}'$ indeed accepts exactly those configurations of $\mathcal{P}$ which violate φ. Moreover, the size of $\mathcal{A}'$ and the time and space bounds to compute it are the same as for $\mathcal{A}$ which proves the first part of the theorem.

To solve problem (V), one can try out the same strategies as in Theorem 6.1. Again, it turns out that the most efficient way is to synchronize $\mathcal{A}'$ with the $\mathcal{P}$-automaton $\mathcal{R}$ which recognizes all reachable configurations of $\mathcal{P}$. Employing the same trick as in Theorem 6.1 (i.e., sorting transitions of $\mathcal{R}$ into buckets according to their labels), we obtain that the size of the synchronized automaton is $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2 \cdot |States| \cdot |\mathcal{B}|^2)$ and it can be computed in $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2 \cdot |States| \cdot |\mathcal{B}|^3)$ time using $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2 \cdot |States| \cdot |\mathcal{B}|^2)$ space. Using the alternative method (sorting transitions of $\mathcal{A}'$ into buckets instead and exploiting determinism) we get an automaton of size $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + |n_2|) \cdot |States| \cdot |\mathcal{B}|^2)$ in $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + |n_2|) \cdot |States| \cdot |\mathcal{B}|^3)$ in time and $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + |n_2|) \cdot |States| \cdot |\mathcal{B}|^2)$ space. $\square$

Incidentally, none of the techniques would work (even with worse complexity bounds) if we drop the requirement that the $\mathcal{M}^p_{A_i}$ automata be deterministic. For instance, checking the formula $\Diamond\varphi$ could fail if a valuation automaton can choose between an accepting and a non-accepting state, thus permitting a run in which no configuration seems to satisfy φ.

Both solutions can be combined with the concept of symbolic pushdown systems in Section 3.3. Since both techniques take the same asymptotic time it would be interesting to compare their efficiency in practice (for cases where both techniques can be used).

6.4 Interprocedural Data-Flow Analysis

In Section 2.3 we discussed how pushdown systems can serve as a model for the behaviour of a sequential program with procedures. In this model, the program is translated into a pushdown system in such a way that each control point of the program corresponds to one particular stack symbol. The topmost stack symbol of a configuration in the pushdown system corresponds to the current program point (and to the instruction which is to be executed), and the stack carries information about the history of procedure calls.

Hence, efficient analysis techniques for pushdown automata can be applied to some problems of interprocedural data-flow analysis [19, 39]. Many of the well-known properties of data-flow analysis can be expressed in LTL and verified by a model-checker. For instance, in Section 2.5 we argued that the formula

$$(\neg used_Y \,\mathcal{U}\, def_Y) \;\vee\; (\Box\neg used_Y)$$

can express that a program variable Y is dead. Information like this is used in program optimization since the values of dead variables may be 'forgotten' by the program. Regular valuations become useful even in this simple example – if we have a language with dynamic scoping (e.g., LISP), we *cannot* resolve to which Y the instruction $Y := 3$ at a program point n refers to without examining the stack of activation records (the Y refers to a local variable Y of the topmost procedure in the stack of activation records which declares it). Hence, $used_Y$ and def_Y would be interpreted by regular valuations in this case.

6.5 Pushdown Systems with Checkpoints

Another area where regular valuations find a natural application is the analysis of recursive computations with local security checks. Modern programming languages contain methods for performing run-time inspections of the stack of activation records, and processes can thus take decisions based on the stack contents. An example is the class `AccessController` implemented in Java Development Kit 1.2 which offers the method `checkPermission` to check whether all methods stored in the stack are granted a given permission. If not, the method rises an exception.

Here, we discuss a formal model of such systems called *pushdown systems with checkpoints*, inspired by Jensen et al [26] who deal with the same problem. The model presented here is more general, however. The model of [26] is suitable only for checking safety properties, does not model data-flow, and forbids mutually recursive procedure calls, whereas our model has none of these restrictions. Properties of pushdown systems with checkpoints can be expressed in LTL and we provide an efficient model-checking algorithm for LTL with regular valuations.

Definition 6.3 *A triple $\mathcal{C} = (\mathcal{P}, \xi, \eta)$ is a* pushdown system with checkpoints *if*

- $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ *is a pushdown system.*
- $\xi \subseteq P \times \Gamma$ *is a set of* checkpoints. *Each checkpoint* (p, α) *is* implemented *by its associated deterministic finite-state automaton* $\mathcal{M}^p_\alpha = (Q^p_\alpha, \Gamma, \rightarrow_{p,\gamma}, s^p_\gamma, F^p_\gamma)$. *For technical convenience, we assume that* $\rightarrow_{p,\gamma}$ *is total,* $s^p_\gamma \notin F^p_\gamma$*, and* $L(\mathcal{M}^p_\gamma) \subseteq \{\, w\gamma \mid w \in \Gamma^* \,\}$.
- $\eta \colon \Delta \rightarrow \{+, -, 0\}$ *is a function which partitions the set of transition rules into* positive, negative, *and* independent *ones. We require that if* (p, γ) *is* not *a checkpoint, then all rules of the form* $\langle p, \gamma\rangle \hookrightarrow \langle p', v\rangle$ *are independent.*

The function η determines whether a rule can be applied when an inspection of the stack at a checkpoint yields a positive or negative result, or whether it is independent of such tests. Using positive and negative rules, we can model systems which perform `if-then-else` commands where the condition is based on dynamic checks; hence, these checks can be nested to an arbitrary level. The fact that a rule $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ is positive, negative, or independent is denoted by $\langle p, \gamma\rangle \hookrightarrow^+ \langle p', w\rangle$, $\langle p, \gamma\rangle \hookrightarrow^- \langle p', w\rangle$, or $\langle p, \gamma\rangle \hookrightarrow^0 \langle p', w\rangle$, respectively.

To $\mathcal{C}$ we associate a unique transition system $\mathcal{T}_\mathcal{C} = (\mathit{Conf}(\mathcal{P}), \rightarrow, c_0)$ whose set of states is $\mathit{Conf}(\mathcal{P})$, whose root is c_0, and whose transition relation is the least relation $\Rightarrow$ satisfying the following:

- if $\langle p, \gamma\rangle \hookrightarrow^+ \langle p', v\rangle$ and $w^R\gamma \in L(\mathcal{M}^p_\gamma)$, then $\langle p, \gamma w\rangle \Rightarrow \langle p', vw\rangle$;
- if $\langle p, \gamma\rangle \hookrightarrow^- \langle p', v\rangle$ and $w^R\gamma \notin L(\mathcal{M}^p_\gamma)$, then $\langle p, \gamma w\rangle \Rightarrow \langle p', vw\rangle$;
- if $\langle p, \gamma\rangle \hookrightarrow^0 \langle p', v\rangle$ and $w \in \Gamma^*$, then $\langle p, \gamma w\rangle \Rightarrow \langle p', vw\rangle$.

Some natural problems for pushdown processes with checkpoints are listed below.

- The reachability problem: given a pushdown system with checkpoints, is a given configuration reachable?
- The checkpoint-redundancy problem: given a pushdown system with checkpoints and a checkpoint (p, γ), is there a reachable configuration where the checkpoint (p, γ) is (or is not) satisfied?

  This problem is important because redundant checkpoints can be safely removed together with all negative (or positive) rules, declaring all remaining rules as independent. Thus, one can decrease the runtime overhead.
- The global safety problem: given a pushdown system with checkpoints and a formula φ of LTL, do all reachable configurations satisfy φ?

  An efficient solution to this problem allows to make experiments with checkpoints with the aim of finding a solution with a minimal runtime overhead.

It is quite easy to see that all these problems (and others) can be encoded by LTL formulas and regular valuations. For example, to solve the reachability problem, we take a predicate A which is satisfied only by the configuration $\langle p, w\rangle$ whose reachability is in question (the associated automaton $\mathcal{M}_A^p$ has $|w| + 1$ states) and then check the formula $\Box(\neg A)$. This formula in fact says that $\langle p, w\rangle$ is *unreachable*; reachability itself is not directly expressible in LTL (we can only say that $\langle p, w\rangle$ is reachable in *every run*). However, it does not matter because we can simply negate the answer of the model-checking algorithm.

In the following we show how to model-check LTL for pushdown systems with checkpoints. Let $\mathcal{C} = (\mathcal{P}, \xi, \eta)$ be a pushdown system with checkpoints, where $\mathcal{P}$ has the form $(P, \Gamma, \Delta, \langle p_0, w_0\rangle)$. We define a pushdown system $\mathcal{P}' = (P \times \{+, -, 0\}, \Gamma, \Delta', \langle (p_0, 0), w_0\rangle)$ where Δ' is the least set of rules satisfying the following (for each $x \in \{+, -, 0\}$);

- if $\langle p, \gamma\rangle \hookrightarrow^+ \langle (p', +), w\rangle \in \Delta$, then $\langle (p, x), \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta'$;
- if $\langle p, \gamma\rangle \hookrightarrow^- \langle (p', -), w\rangle \in \Delta$, then $\langle (p, x), \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta'$;
- if $\langle p, \gamma\rangle \hookrightarrow^0 \langle (p', 0), w\rangle \in \Delta$, then $\langle (p, x), \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta'$.

Intuitively, $\mathcal{P}'$ behaves in the same way as the underlying pushdown system $\mathcal{P}$ of $\mathcal{C}$, but it also remembers which kind of rule (positive, negative, independent) was used to enter the current configuration. However, we need to exclude runs in which $\mathcal{P}'$ 'cheats' by executing a positive/negative rule when the associated checkpoint condition is violated/satisfied.

Let φ be an LTL formula, and let ν be a regular valuation for $At(\varphi)$ on configurations of $\mathcal{C}$ (see Definition 6.1). Let *Check*, *Neg*, and *Pos* be fresh atomic propositions which do not appear in φ. We define a regular valuation ν' for configurations of $\mathcal{P}'$ as follows:

- If $A \in At(\varphi)$, let $\nu'(A) = \{\, \langle (p,x), w\rangle \mid \langle p, w\rangle \in \nu(A),\ x \in \{+,-,0\}\,\}$. Hence, the automaton $\mathcal{M}_A^{(p,x)}$ is the same as the automaton $\mathcal{M}_A^p$ for each $x \in \{+,-,0\}$.
- $\nu'(\mathit{Check}) = \bigcup_{(p,\gamma)\in\xi}\{\, \langle (p,x), \gamma w\rangle \mid w^R\gamma \in L(\mathcal{M}_\gamma^p),\ x \in \{+,-,0\}\,\}$. Hence, for each $x \in \{+,-,0\}$, $\mathcal{M}_{\mathit{Check}}^{(p,x)}$ is the product automaton (constructed out of all $\mathcal{M}_\gamma^p$) which accepts the union of all $L(\mathcal{M}_\gamma^p)$. Notice that we need to compute a product because $\mathcal{M}_{\mathit{Check}}^{(p,x)}$ has to be deterministic.
- $\nu'(\mathit{Neg}) = \{\, \langle (p,-), w\rangle \mid p \in P,\ w \in \Gamma^+\,\}$. Hence, $\mathcal{M}_{\mathit{Neg}}^{(p,-)}$ is an automaton with two states which accepts Γ^+.
- $\nu'(\mathit{Pos}) = \{\, \langle (p,+), w\rangle \mid p \in P,\ w \in \Gamma^+\,\}$.

Now we can readily confirm the following:

Theorem 6.3 *Let $\langle p, w\rangle$ be a configuration of $\mathcal{C}$. We have that*

$$\langle p, w\rangle \models^{\nu} \varphi \qquad\Longleftrightarrow\qquad \langle (p,0), w\rangle \models^{\nu'} (\psi \Longrightarrow \varphi)$$

where $\psi \equiv \Box((\mathit{Check} \Longrightarrow \mathcal{X}(\neg \mathit{Neg})) \wedge (\neg \mathit{Check} \Longrightarrow \mathcal{X}(\neg \mathit{Pos})))$.

Proof: It suffices to observe that

$$\langle p, w\rangle = \langle p_1, w_1\rangle \to \langle p_2, w_2\rangle \to \langle p_3, w_3\rangle \to \cdots$$

is an infinite path in $\mathcal{T}_{\mathcal{C}}$ if and only if

$$\langle (p,0), w\rangle = \langle (p_1,x_1), w_1\rangle \to \langle (p_2,x_2), w_2\rangle \to \langle (p_3,x_3), w_3\rangle \to \cdots$$

is an infinite path in $\mathcal{T}_{\mathcal{P}'}$ satisfying ψ. Indeed, ψ ensures that all transitions in the latter path are consistent with possible checkpoints in the former path. As all atomic propositions which appear in φ are evaluated identically for pairs $\langle p_i, w_i \rangle$ and $\langle (p_i, x_i), w_i \rangle$ (see the definition of ν' above), we conclude that both paths either satisfy or do not satisfy φ. $\square$

The previous theorem in fact says that the model-checking problem for LTL and pushdown systems with checkpoints can be reduced to the model-checking problem for LTL and 'ordinary' pushdown systems. As the formula ψ is fixed and the atomic propositions *Check*, *Neg*, and *Pos* are regular, we can evaluate the complexity bounds for the resulting model-checking algorithm using Theorems 6.1 and 6.2. Let $At(\varphi) = \{A_1, \ldots, A_n\}$, and let $\mathcal{N} = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ be the set of all $\mathcal{M}^p_{A_i}$ automata. Let *States* be the Cartesian product of the sets of states of all $\mathcal{M}^p_\gamma$ automata and the automata of $\mathcal{N}$. Let $\mathcal{B}$ be a Büchi automaton which corresponds to $\neg\varphi$. Let $\mathcal{P} = (P, \Gamma, \Delta, \langle p_0, w_0 \rangle)$ be the underlying pushdown system of $\mathcal{C}$, and let n_2 be the number of different pairs $\langle p', \gamma \rangle$ such that Δ has a rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$. Now we can state the theorem:

Theorem 6.4 *We have the following bounds on the model-checking problems for LTL with regular valuations and pushdown systems with checkpoints:*

1. *Problems (III) and (IV) can be solved in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot |States| \cdot |\mathcal{B}|^2)$ space.*
2. *Problem (V) can be solved in either $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2 \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot (|P| + |w_0| + n_2)^2 \cdot |States| \cdot |\mathcal{B}|^2)$ space, or $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + |n_2|) \cdot |States| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P|^3 \cdot |\Delta| \cdot (|w_0| + |n_2|) \cdot |States| \cdot |\mathcal{B}|^2)$ space.*

Proof: We apply Theorem 6.3, which says that we can instead consider the problem for the pushdown system $\mathcal{P}'$, the formula $\psi \implies \varphi$, and the valuation ν'. The Büchi automaton which corresponds to $\neg(\psi \implies \varphi)$ can be obtained by synchronizing $\mathcal{B}$ with the Büchi automaton for ψ, because $\neg(\psi \implies \varphi) \equiv (\psi \wedge \neg\varphi)$. As the formula ψ is fixed, the synchronization increases the size of $\mathcal{B}$ just by a constant factor. Hence, the automaton for $\neg(\psi \implies \varphi)$ is asymptotically of the same size as $\mathcal{B}$. The same can be said about the sizes of P' and P, and about the sizes of Δ' and Δ. Moreover, if we collect all the automata which represent the atomic predicates $At(\psi \implies \varphi)$

and consider the state space of their product, we see that it has exactly the size $2 \cdot States$ because all of the automata associated to *Pos* and *Neg* are the same and have only two states. □

6.6 Model-checking CTL*

In this section, we apply the model-checking algorithm to the logic CTL* which extends LTL with existential path quantification [15]. More precisely, CTL* formulas are built according to the following abstract syntax equation:

$$\varphi \ ::= \ \mathtt{tt} \mid A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{E}\varphi \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where A ranges over the atomic propositions (interpreted, say, by a regular valuation represented by a finite automaton of size $|States|$).

We adapt a model-checking procedure from the domain of finite-state systems. In finite-state systems, model-checking CTL* can be reduced to LTL by checking subformulas of φ in ascending order of their nesting depth of existential path quantifiers [16]. The procedure consists of the following iteration:

- Pick any subformula of the form $\mathcal{E}\,\varphi'$, where φ' itself does not contain any $\mathcal{E}$-operators. Then φ' is actually a formula of LTL.
- Apply a model-checking algorithm for LTL on $\neg\varphi'$ which discovers the set of states $S_{\varphi'}$ which violate $\neg\varphi'$. These states are starting points for a run which satisfies φ, hence the states in $S_{\varphi'}$ satisfy $\mathcal{E}\,\varphi'$.
- Replace all occurrences of $\mathcal{E}\,\varphi'$ in φ by a fresh atomic proposition whose valuation is exactly the set of states $S_{\varphi'}$.

The procedure is repeated until φ itself is reduced to a formula without $\mathcal{E}$-operators which can be verified with a model-checking algorithm for LTL.

This method can be transferred to the case of pushdown systems. Notice that the LTL algorithm on an $\mathcal{E}$-free subformula $\neg\varphi'$ returns an automaton $\mathcal{M}_{\varphi'}$ which accepts the regular set of configurations which satisfy $\mathcal{E}\,\varphi'$. We can then replace all occurrences of $\mathcal{E}\,\varphi'$ by a fresh atomic proposition whose valuation is given by $\mathcal{M}_{\varphi'}$.

Let us review the complexity of this procedure. For the rest of this subsection fix a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$. Given an $\mathcal{E}$-free formula φ', let $\mathcal{B} = (R, 2^{At}, \eta, r_0, G)$ be a Büchi automaton corresponding to φ', and let $|States|$ be the size of the $\mathcal{M}^p_{A_i}$ automata which encode the regular valuations of propositions in At.

In general we can only use Technique 2, i.e. extending the stack. Let $\mathcal{M}_\varphi$ be the automaton (obtained by Technique 2) which accepts exactly the configurations satisfying $\mathcal{E}\,\varphi'$. Observe that $\mathcal{M}_\varphi$ is non-deterministic, reads the stack top-down, and has $\mathcal{O}(|P| \cdot |R| \cdot |States|)$ states. We need to modify the automaton before we can use it as an encoding for the regular valuation of $\mathcal{E}\,\varphi$. More precisely, we need to reverse the automaton (i.e. make it read the stack bottom-up) and then determinize it. Reversal does not increase the size, and due to the determinism of $\mathcal{M}^p_{A_i}$ in bottom-up direction the determinization increases only the '$P \times R$ part' of the states. Thus, we get an automaton $\mathcal{M}'_\varphi$ of size $O(|States| \cdot 2^{|P|\cdot|R|})$.

To check subformulas of higher nesting depth (of $\mathcal{E}$-operators) we replace $\mathcal{E}\,\varphi$ by a fresh atomic proposition A_φ. With (A_φ, p) we associate the automaton $\mathcal{M}^p_{A_\varphi}$ which is a copy of $\mathcal{M}'_\varphi$ where the sets of accepting states are defined as

$$F^p_\varphi = \{\, (q, \mathbf{s}) \mid q \in 2^{P\times R},\ (p, r_0) \in q,\ \mathbf{s} \in States \,\}.$$

The cross product of these new automata with the 'old' $\mathcal{M}^p_{A_i}$ automata again has the size $O(|States| \cdot 2^{|P|\cdot|R|})$, since we need just one copy of the new automaton, and all reachable states are of the form $((q, \mathbf{s}), \mathbf{s})$ where $q \in 2^{P\times R}$ and $\mathbf{s} \in States$.

As nesting depth increases, we repeat the procedure: First we produce a deterministic valuation automaton by taking the cross product of the automata for the atomic propositions and those derived from model-checking formulas of lower nesting depth. Then we model-check the subformula currently under consideration, and reverse and determinize the resulting automaton. By the previous arguments, each determinization only blows up the non-deterministic part of the automaton. Therefore, after each stage the size of the valuation automaton increases by a factor of $2^{|P|\cdot|\mathcal{B}_i|}$ where $\mathcal{B}_i$ is a Büchi automaton for the subformula under consideration.

With this in mind, we can compute the complexity for formulas of arbitrary nesting depth. Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be the Büchi automata corresponding to the individual subformulas of a formula φ. Adding the times for checking the subformulas and using Theorem 6.2 we get that the model-checking procedure takes at most

$$\mathcal{O}\Bigg(|P|^2 \cdot |\Delta| \cdot |States| \cdot 2^{|P| \cdot \Sigma_{i=1}^{n} |\mathcal{B}_i|} \cdot \sum_{i=1}^{n} |\mathcal{B}_i|^3\Bigg)$$

time and

$$\mathcal{O}\Bigg(|P| \cdot |\Delta| \cdot |States| \cdot 2^{|P| \cdot \Sigma_{i=1}^{n} |\mathcal{B}_i|} \cdot \sum_{i=1}^{n} |\mathcal{B}_i|^2\Bigg)$$

space. The algorithm hence remains linear in both $|\Delta|$ and $|States|$.

The essential ideas contained in this method were already proposed by Finkel et al in [22], but without any complexity analysis. Also, the presentation in [22] did not discuss the issue of having to use *deterministic* automata for the valuations.

The algorithm of Burkart and Steffen [12] for the μ-calculus on pushdown systems, applied to CTL* formulas which are in the second level of the alternation hierarchy, would yield an algorithm which is cubic in $|\Delta|$. On the other hand, the performance of the two algorithms in terms of the formula is less clear since their techniques differ in an interesting way. In [12], the exponential blowup occurs because configurations are extended with a 'context' which indicates the subformulas which will be true once the topmost symbol is removed from the stack. In our case, the exponential blowup occurs because we need to determinize finite automata. However, these automata are derived from Büchi automata which are usually constructed by a tableau method that labels states with certain subformulas. Therefore, the complexity estimations of the two algorithms might be related in a subtle fashion which would be interesting to examine.

More recently, Benedikt et al [6] have proposed a method for model-checking CTL* on hierarchical state machines (see Section 3.1.5). Their algorithm is also linear in the number of edges whose size corresponds to Δ; however, it works only for single-exit state machines whose expressive power is equal to that of pushdown systems with only one control location.

6.7 Lower Bounds

In previous sections we established upper bounds for the model-checking problem for pushdown systems (with and without checkpoints) and LTL with regular valuations. The algorithms are polynomial in $|\mathcal{P}| + |\mathcal{B}| + |States|$, but not in $|\mathcal{P}| + |\varphi| + |\nu_\varphi|$, which can be regarded as the real size of the problem instance (see Remark 6.1). We also discussed *why* it makes sense to use these parameters; typical formulas and their associated Büchi automata are small, and the size of $\mathcal{B}$ is actually more relevant (a model-checking algorithm whose complexity is exponential *just* due to the blowup caused by the transformation of φ into $\mathcal{B}$ is usually efficient in practice). The same can be said about $|States|$ – in Sections 6.4 and 6.5 we have seen that there are interesting practical problems where the size of $|States|$ does not explode. Nevertheless, from the point of view of worst-case analysis, where the complexity is measured in the size of the problem instance, the algorithms are *exponential*. A natural question is whether this exponential blowup is indeed necessary, i.e., whether we could in principle solve the model-checking problems more efficiently by some other technique. However, one can show that this is *not* the case, because all of the considered problems are **EXPTIME**-hard, even in rather restricted forms.

At this point, we only state the results and briefly discuss their relevance. The technical proofs of the theorems can be found in [20].

We start with the natural problems for pushdown systems with checkpoints mentioned in the previous section (the reachability problem, the checkpoint redundancy problem, etc). All of them are polynomially reducible to the model-checking problem for pushdown systems with checkpoints and LTL with regular valuations and therefore are solvable in **EXPTIME**. Theorem 6.5 says that this strategy is essentially optimal, because even the reachability problem provably requires exponential time.

Theorem 6.5 *The reachability problem for pushdown systems with checkpoints (even with just three control states and without negative rules) is* **EXPTIME**-*complete.*

From Theorem 6.5 we can easily deduce the following result for LTL:

Theorem 6.6 *The model-checking problem (III) for pushdown systems with checkpoints (even with just three control states and no negative rules) is* ***EXPTIME****-complete even for a fixed LTL formula* $\Box(\neg \mathit{fin})$ *where fin is an atomic predicate interpreted by a* simple *valuation* ν*.*

It follows that model-checking LTL for pushdown systems with checkpoints is **EXPTIME**-complete in the size of the system even when we have only *simple* valuations (recall that the problem is polynomial for systems without checkpoints).

Finally, we consider the complexity of model-checking for (ordinary) pushdown systems and LTL formulas with regular valuations. First, realize that if we take any fixed formula and a subclass of pushdown systems where the number of control states is bounded by some constant, the model-checking problem is decidable in *polynomial* time. Theorem 6.7 says that if the number of control states is not bounded, the model-checking problem becomes **EXPTIME**-complete even for a fixed formula.

Theorem 6.7 *The model-checking problem (III) for pushdown systems and LTL formulas with regular valuations is **EXPTIME**-complete even for a fixed formula* $(\Box\, \mathit{correct}) \Longrightarrow (\Box \neg \mathit{fin})$.

Chapter 7

Conclusion

The main contribution of the thesis is the development of model-checking algorithms for pushdown systems both in theory and in practice. The problems we have considered were already known to be decidable in polynomial time [9, 22], but no implementation for these solutions was available. The algorithms developed herein improve the previously known bounds, and an implementation has been provided. The implementation has been successfully tested on large examples taken from an industrial setting, and the tests have provided insights into practical aspects of designing a model-checker for software verification.

The work can be continued and improved in several aspects. For reachability checking on Boolean Programs, Section 4.2.3 suggests some concrete improvements which are likely to improve performance even further. Moreover, one could extend the existing implementation to other types of symbolic representations besides BDDs. For instance, Number Decision Diagrams [8] are used to represent finite and infinite sets of integer vectors; it would be interesting to see whether they can be used more efficiently than BDDs in examples which use integer variables (like Quicksort etc).

One weakness of pushdown systems is that they do not allow to model parallelism. Linear-time temporal logic becomes undecidable for systems which allow both parallel composition and procedure calls [31]. For reachability, the addition of parallel composition (even without synchronization) lifts the associated problems into different complexity classes. Polynomial reachability algorithms exist for PA-processes [21], which could be characterized as pushdown systems with parallel composition, but with only one control location. However, if the latter restriction is lifted, reachability becomes NP-complete, and even EXPSPACE-hard if synchronization between processes is also allowed [31]. Therefore, it is not straightforward to extend the algorithms for pushdown systems with parallel composition (and maintain efficiency at the same time). It would be worthwhile to try and identify useful special cases and limitations which allow to use parallel composition while maintaining efficiency in practice (or decidability, in the case of LTL).

Finally, in a recent development a new application for pushdown systems has been proposed. Jha and Reps [27] suggest that model-checking pushdown systems can be used to solve a number of certificate-analysis problems in authorization. In this context, certificates grant permissions to access certain resources and define principals who may apply for permissions. A set of certificates can be interpreted as a pushdown system, and so algorithms for the latter can be used to answer questions such as "Is a given principal allowed to access a given resource?" or "Do all authorizations for a given resource involve a certificate signed by a certain principal?" Using extensions similar to shortest traces, one can also solve 'quantified' problems such as "How long does a specific authorization last?" or "How trustworthy is a given authorization?" Exploring these matters in more detail promises to be an interesting line of future work.

Appendix A

Moped tool description

This appendix describes some technical details of the Moped tool, i.e. the languages used for specifying systems and formulas, and options which the user can pass to the tool to influence its behaviour.

Section A.1 describes the syntax and semantics of the language used for specifying symbolic pushdown systems. Section A.2 gives the syntax of Boolean Programs. The format and meaning of reachability queries and LTL formulas is shown in Section A.3, and Section A.4 lists the options.

A.1 Syntax for pushdown systems

Formally, we defined a pushdown system as a quadruple (P, Γ, Δ, c_0). For pushdown systems in Moped, P always has the form $P_0 \times G$ and Γ always has the form $\Gamma_0 \times L$, where G and L are domains of global and local variables, respectively. To describe such a system, it is only necessary to define the rules Δ and the initial configuration c_0; the sets P_0 and Γ_0 are defined implicitly, i.e. whenever an identifier occurs in a place where an element of P_0 or Γ_0 is expected, then the identifier is included in the appropriate set. To define the syntax of the input language more precisely, we use the following conventions:

- Keywords are `global`, `local`, `bool`, `int`, `define`, `A`, and `E`.

- An *identifier* is any alphanumeric string, i.e. a letter followed by a sequence of letters and digits (where the underscore _ counts as a letter), except for keywords. Case is significant.

- A *string* is a sequence of characters other than quotes (`"`) and newlines which are surrounded by quotes.
- A *number* is any non-empty sequence of digits.
- Whitespace characters are newlines, spaces and tab characters and are ignored (except in *string*s and insofar as they separate *identifier*s and *number*s).
- Any string starting with a percent sign (`%`) or a hash character (`#`) (unless these appear in a *string*) and ending with a newline is a comment and is ignored.
- Any keywords and other recognized tokens are enclosed in quotes in the syntax description below.
- We adopt the following grammatic meta-rules: If x is a non-terminal, then x*–list* is a sequence of zero or more occurrences of x, and x*–clist* is a sequence of one or more comma-separated occurrences of x.

A description of a pushdown system consists of four sections: Definitions of constants, variable declarations, the initial configuration, and the rewrite rules:

pds : :
  definition–list vardecl initconfig rule–list

We use two grammar rules for identifiers simply to make clear that the identifiers in question stand for control locations and stack symbols, respectively.

ctrlid : :
  identifier

stackid : :
  identifier

Definitions of constants Each *definition* establishes a symbolic integer constant.

definition : :
  `define` *identifier constexpr*

Note: A definition is *ignored* if the *identifier* in question has already been the subject of an earlier constant definition, either by a *definition* or by passing a `-D` option to Moped(see Section A.4). Once the definition has been read, any occurrence of the newly defined *identifier* is in fact treated as an occurrence of the *number* represented by the *constexpr* part. A *constexpr* is an expression consisting of integer constants and basic arithmetic operators.

constexpr `::`
  constexpr `"+"` *constexpr*
| *constexpr* `"-"` *constexpr*
| *constexpr* `"*"` *constexpr*
| *constexpr* `"/"` *constexpr*
| *constexpr* `"<<"` *constexpr*
| `"("` *constexpr* `")"`
| *number*

Variable declarations The declarations consist of a global and a local part:

vardecl `::`
  globaldecl localdecl–list

The global declarations are either empty or a list of variable declarations:

globaldecl `::`
  ε
| `global` *var–list*

The domain of global variables is spanned by the variables in *var–list* (see below), or a singleton set if omitted. A *localdecl* also includes a list of identifiers:

localdecl `::`
  `local "("` *stackid–clist* `")"` *var–list*

Each *stackid* is interpreted as an explicit stack symbol whose domain of local variables is given by *var–list*. It is illegal for a stack symbol to occur in two different *localdecl* elements.

Variables can be of type boolean or integer, or arrays thereof:

```
var ::
        bool boolident–clist ";"
      | int intident–clist ";"

boolident ::
        identifier optdim

intident ::
        identifier optdim "(" constexpr ")"

optdim ::
        ε
      | "[" constexpr "]"
      | "[" constexpr "," constexpr "]"
```

The range of each integer variable is given by the integer constant in parentheses; if that constant evaluates to k, then the range is $0 \ldots 2^k - 1$. If *optdim* is empty, then the variable in question is scalar, otherwise it is an array. If the brackets contain a single *constexpr* of value m, then the array has m entries indexed by $0 \ldots m-1$. If two values m and n are given, then we expect $n \geq m$, and the array has $n - m + 1$ entries indexed by $m \ldots n$.

Initial configuration As initial configurations, the input language allows configurations with a single stack symbol:

```
initconfig ::
        "(" ctrlid "<" stackid ">" ")"

optexpr ::
        ε
      | "(" expr ")"
```

Although having just one initial stack symbol may seem more restrictive at first sight compared to the usual definition, it does not really lose any generality. One could actually simulate even regular sets of initial configurations by starting with an 'artificial' initial configuration with just one symbol, and introducing additional rules to simulate a finite automaton, which builds up the possible stack contents before the 'real' pushdown system starts. The initial values of the global and local variables can be any value of G or L, respectively.

Rewrite rules Each *rule* defines a labelled symbolic rewrite rule.

rule : :
  explicitpart optlabel optexpr

optlabel : :
  ε
| *string*

explicitpart : :
  ctrlid "<" *stackid* ">" "-->" *ctrlid* "<" ">"
| *ctrlid* "<" *stackid* ">" "-->" *ctrlid* "<" *stackid* ">"
| *ctrlid* "<" *stackid* ">" "-->" *ctrlid* "<" *stackid stackid* ">"

Thus, the stack contents on the right-hand side are limited to length two. The label is optional and only used in printing witnesses or counterexamples. The *optexpr* is a boolean expression which denotes a relation of global and local variables:

optexpr : :
  ε
| "(" *expr* ")"

expr : :
  expr "|" *expr*
| *expr* "&" *expr*
| *expr* "==" *expr*
| *expr* "^" *expr*
| "!" *expr*
| "(" *expr* ")"
| `A` *quant expr*
| `E` *quant expr*
| *variable*
| *intterm compop intterm*

The first couple of branches represent the usual logical connectives. The rules with `A` and `E` are universal and existential quantification:

quant : :
  identifier "(" *constexpr* "," *constexpr* ")"

An `A/E` rule corresponds to conjunction/disjunction of the terms which are obtained by replacing every occurrence of *identifier* in *expr* with each of the values between (and including) the two constants in turn. The remaining three rules govern the use of boolean and integer variables. A *variable* is an *identifier* followed by zero, one, or two primes and optionally followed by an index enclosed in brackets:

variable : :
    identifier primes optindex

primes : :
    ε| "'" | "''"

optindex : :
    ε
  | "[" *intterm* "]"

A *variable* can occur in two places; directly in place of an *expr*, or inside an *intterm* (see below). In the first case its *identifier* must be a global or local variable declared as `bool`. It is an error for a scalar variable to be followed by an index, or for an array variable not to. Unprimed identifiers refer to the global and local variables on the left-hand side, singly primed identifiers to the right-hand side control location and the first stack symbol (if present), and doubly primed identifiers to the second stack symbol (if present). A local variable may occur in an expression with zero (one, two) primes if it is in the domain associated with the left-hand side (first, second right-hand side) explicit stack symbol. The last rule for *expr* allows two integer-valued terms to be compared by the usual arithmetic operators:

intterm : :
    intterm "+" *intterm*
  | *intterm* "-" *intterm*
  | *intterm* "*" *intterm*
  | *intterm* "/" *intterm*
  | *intterm* "<<" *intterm*
  | *variable*
  | *quantifier*
  | *number*

quantifier : :

identifier

compop ::
```
"<" | "<=" | "=" | "!=" | ">=" | ">"
```

The *identifier* of a *variable* inside an *intterm* must be of type `int`. The rules for scalars and arrays are the same as for boolean variables. An *identifier* used in place of a *quantifier* must have been declared in an `A`- or `E`-clause which includes the *intterm* it occurs in.

All binary operators which occur in *expr*, *intterm*, and *constexpr* clauses associate to the left. The order of precedence is as follows, from highest to lowest:

```
"<<"
"*", "/"
"+", "-"
"!"
"&"
"|"
"^"
"=="
A, E
```

Semantics We now give the precise semantics of the input language. All occurrences of *constexpr* can be evaluated statically to an integer constant. The operators `+`,`-`,`*`,`/` have their usual arithmetic meanings, and `<<` stands for "shift left", i.e. m `<<` $n = m \cdot 2^n$. In the following, we represent each *constexpr* by the constant it evaluates to.

We give the semantics in terms of a symbolic pushdown system; in doing so, we deviate a little from the usual definition by defining a *set* of initial configurations C_0. However, this does not make any significant difference in practice. The symbolic pushdown system defined by the input has the form

$$(P_0 \times G, \Gamma_0 \times L, \Delta_S, C_0)$$

in which P_0 is the set of all *identifier*s which occur in place of a *ctrlid*, and Γ_0 is the set of all *identifier*s which occur in place of a *stackid*. The sets G and L, depend on the contents of the *vardecl* part. Let us associate a 'size' with each declaration which represents the number of bits needed for each variable.

- A declaration of the form "`bool x;`" creates a scalar boolean variable and has size 1. In the following, we set $\mathit{range}(\mathtt{x}) := \{0,1\}$.
- A declaration of the form "`bool x[`m`];`" creates an array of boolean variables with indices $0 \ldots m-1$; its size is m. likewise, a declaration "`bool x[`m,n`];`" creates an array with indices $m,\ldots,n$, and the size is $n-m+1$. Again, we set $\mathit{range}(\mathtt{x}) := \{0,1\}$.
- A declaration of the form "`int x(`k`);`" creates a scalar integer variable with $\mathit{range}(\mathtt{x}) := \{0,\ldots,2^k-1\}$, and the size is k.
- A declaration of the form "`int x[`m`](`k`);`" creates an array of integers with $\mathit{range}(\mathtt{x}) := \{0,\ldots,2^k-1\}$; the array has indices $0,\ldots,m-1$ and the size is $m \cdot k$. In the case "`int x[`m,n`](`k`);`" the indices are $m,\ldots,n$ and the size $(n-m+1)\cdot k$.

If a *boolident–clist* or *intident–clist* contains more than one element, we treat them as several individual declarations of the same type; allowing multiple elements in the list merely adds syntactical convenience. Now, let m be the result of summing up the sizes of all declarations in *globaldecl*, or 0 if *globaldecl* is empty. We set $G := \{0,1\}^m$. Moreover, we can construct a bijection f_g between G and the valuations of the global variables. For the presentation of the semantics, we introduce for every scalar global variable x a function

$$\langle \mathtt{x} \rangle^{\mathrm{s}}_{\mathrm{G}} \colon G \to \mathit{range}(\mathtt{x})$$

which extracts the value of `x` from the valuation corresponding to its argument. Analogously, we use a function $\langle \mathtt{x} \rangle^{i}_{\mathrm{G}}$ for every global array `x` and every index i of `x`. If we have a *localdecl* of the form

`local "(" ` *stackid–clist* ` ")" ` *var–list*

then we set $\mathit{dom}(\gamma)$ to the set of all variables declared in *var–list* and $\mathit{size}(\gamma)$ to the sum of the sizes of their declarations, for all $\gamma \in \Gamma_0$ which occur in the *stackid–clist* (we assume $\mathit{dom}(\gamma) = \emptyset$ and $\mathit{size}(\gamma) = 0$ for all $\gamma \in \Gamma_0$ which do not occur in any *localdecl*). If $n = \max\{\, \mathit{size}(\gamma) \mid \gamma \in \Gamma_0 \,\}$, then we set $L := \{0,1\}^n$. It is then obvious that for every $\gamma \in \Gamma_0$ we can construct a surjective function f_γ from L into the valuations of the variables in $\mathit{dom}(\gamma)$. Like for the global variables, we introduce functions $\langle \mathtt{x} \rangle^{\mathrm{s}}_{\gamma}$ for every scalar $\mathtt{x} \in \mathit{dom}(\gamma)$ and $\langle \mathtt{x} \rangle^{i}_{\gamma}$ for every index i of an array $\mathtt{x} \in \mathit{dom}(\gamma)$ to extract the value of `x` (or $\mathtt{x}[i]$) from the valuations which correspond to elements of L.

The precise construction of f_g, f_γ (and hence $\langle\cdot\rangle$) is actually flexible and is not discussed here. In the case of BDDs, this issue boils down to the questions how BDD variables should be allocated to variables in the pushdown systems, and which ordering should be used on the BDD variables. These issues are discussed to some extent in Chapter 5.

We are now in a position to give the semantics of rewrite rules. Even though the semantics may look complicated at first, it is actually fairly natural, and all operations on boolean expressions and variables correspond directly to certain BDD operations.

For every *rule* in the parse tree whose *explicitpart* has the form

$$p_1 \ \texttt{<}\gamma\texttt{>} \ \texttt{-->} \ p_2 \ \texttt{<}w\texttt{>}$$

where w contains the *stackid*s on the right, Δ_S contains a rewrite rule

$$\langle p_1, \gamma\rangle \xhookrightarrow[{[R]}]{} \langle p_2, w\rangle$$

where $R = G \times L \times G \times L^{|w|}$ if the corresponding *optexpr* is empty. Otherwise we determine R by means of three functions $[\![\cdot]\!]$, $\langle\!\langle\cdot\rangle\!\rangle$, and *val*. $[\![\cdot]\!]$ maps *expr*s to sets of tuples, $\langle\!\langle\cdot\rangle\!\rangle$ evaluates *intterm*s, and *val* extracts values from tuples. We start with the latter:

1. For every *variable* v which occurs in the parse tree, let $\mathtt{x}_v$ be its *identifier*, p_v the number of primes, and t_v the *intterm* inside its *optindex* part if the latter is not empty. Given a tuple $h \in G \times L \times G \times L^{|w|}$ and a 'substitution set' S, we define $val^S_{\gamma w}(v)(h)$ as $\langle \mathtt{x}_v\rangle^i_d(k)$, where

$$i = \begin{cases} \mathrm{s} & \text{if } \mathtt{x}_v \text{ is scalar} \\ \langle\!\langle t_v\rangle\!\rangle^S_{\gamma w}(h) & \text{if } \mathtt{x}_v \text{ is an array} \end{cases}$$

$$d = \begin{cases} \mathrm{G} & \text{if } \mathtt{x}_v \text{ is global} \\ \gamma & \text{if } p_v = 0,\ \mathtt{x}_v \in \mathit{dom}(\gamma) \\ \gamma_1 & \text{if } p_v = 1,\ |w| \geq 1,\ w = \gamma_1 w',\ \mathtt{x}_v \in \mathit{dom}(\gamma_1) \\ \gamma_2 & \text{if } p_v = 2,\ |w| = 2,\ w = \gamma_1\gamma_2,\ \mathtt{x}_v \in \mathit{dom}(\gamma_2) \end{cases}$$

$$k = \begin{cases} g & \text{if } \mathtt{x}_v \text{ is global, } p_v = 0,\ h = (g, l, g', \ldots) \\ g' & \text{if } \mathtt{x}_v \text{ is global, } p_v = 1,\ h = (g, l, g', \ldots) \\ l & \text{if } p_v = 0,\ \mathtt{x}_v \in \mathit{dom}(\gamma),\ h = (g, l, g', \ldots) \\ l' & \text{if } p_v = 1,\ |w| \geq 1,\ w = \gamma_1 w',\ \mathtt{x}_v \in \mathit{dom}(\gamma_1),\ h = (g, l, g', l', \ldots) \\ l'' & \text{if } p_v = 2,\ |w| = 2,\ w = \gamma_1\gamma_2,\ \mathtt{x}_v \in \mathit{dom}(\gamma_2),\ h = (g, l, g', l', l'') \end{cases}$$

2. Let t be an *intterm* in the parse tree, h a tuple from $G \times L \times G \times L^{|w|}$, and S a 'substitution set'. We inductively define $\langle\!\langle t \rangle\!\rangle^S_{\gamma w}(h)$ on the structure of t:

   - The arithmetic operators have the same function as in *constexpr*; thus, if *op* is one of `+,-,*,/,<<`, then

   $$\langle\!\langle t_1 \ op \ t_2 \rangle\!\rangle^S_{\gamma w}(h) = \langle\!\langle t_1 \rangle\!\rangle^S_{\gamma w}(h) \ op \ \langle\!\langle t_2 \rangle\!\rangle^S_{\gamma w}(h).$$

   - If t is a *variable*, then $\langle\!\langle t \rangle\!\rangle^S_{\gamma w}(h) = val^S_{\gamma w}(t)(h)$.
   - If t is a *quantifier*, then $\langle\!\langle t \rangle\!\rangle^S_{\gamma w}(h) = i$ if $t/i \in S$.
   - If t is a *number*, then simply $\langle\!\langle t \rangle\!\rangle^S_{\gamma w}(h) = t$.

3. Finally, we can complete the semantics with the $[\![\cdot]\!]$ function, which defines subsets of $G \times L \times G \times L^{|w|}$. Given an *expr* e and a 'substitution set' S, $[\![e]\!]^S_{\gamma w}$ is inductively defined by the following equations:

   - $[\![e_1 \mid e_2]\!]^S_{\gamma w} = [\![e_1]\!]^S_{\gamma w} \cup [\![e_2]\!]^S_{\gamma w}$
   - $[\![e_1 \,\&\, e_2]\!]^S_{\gamma w} = [\![e_1]\!]^S_{\gamma w} \cap [\![e_2]\!]^S_{\gamma w}$
   - $[\![e_1 == e_2]\!]^S_{\gamma w} = ([\![e_1]\!]^S_{\gamma w} \cap [\![e_2]\!]^S_{\gamma w}) \cup ([\![!e_1]\!]^S_{\gamma w} \cap [\![!e_2]\!]^S_{\gamma w})$
   - $[\![e_1 \,\hat{}\, e_2]\!]^S_{\gamma w} = [\![!(e_1 == e_2)]\!]^S_{\gamma w}$
   - $[\![!e]\!]^S_{\gamma w} = (G \times L \times G \times L^{|w|}) \setminus [\![e]\!]^S_{\gamma w}$
   - $[\![(e)]\!]^S_{\gamma w} = [\![e]\!]^S_{\gamma w}$
   - $[\![\mathtt{A}\ \mathtt{q}\ (m,n)\ e]\!]^S_{\gamma w} = \bigcap_{i=m}^{n} [\![e]\!]^{S \cup \{\mathtt{q}/i\}}_{\gamma w}$
   - $[\![\mathtt{E}\ \mathtt{q}\ (m,n)\ e]\!]^S_{\gamma w} = \bigcup_{i=m}^{n} [\![e]\!]^{S \cup \{\mathtt{q}/i\}}_{\gamma w}$
   - If e is of the form *variable*, then

   $$[\![e]\!]^S_{\gamma w} = \{\, h \in G \times L \times G \times L^{|w|} \mid val^S_{\gamma w}(e)(h) = 1 \,\}.$$

   - All comparison operators in *compop* have their usual meanings (`"!="` being the rendering for $\neq$). Thus, if *op* is a *compop*, then

   $$[\![t_1 \ op \ t_2]\!]^S_{\gamma w} = \{\, h \in G \times L \times G \times L^{|w|} \mid \langle\!\langle t_1 \rangle\!\rangle^S_{\gamma w}(h) \ op \ \langle\!\langle t_2 \rangle\!\rangle^S_{\gamma w}(h) \,\}.$$

We set $R := [\![optexpr]\!]^{\emptyset}_{\gamma w}$. If R cannot be evaluated because some subexpression is not defined, then the input contains an error.

Finally, the initial configurations are defined by the *initconfig* part of the parse tree. If p is the *ctrlid* and γ the *stackid* in *initconfig*, the set C_0 of initial configurations is:

$$\{\, \langle (p, g), (\gamma, l) \rangle \mid g \in G,\ l \in L \,\}$$

A.2 Syntax of Boolean Programs

At the time of writing, published descriptions for the syntax of Boolean Program are missing some of its features. Therefore, an up-to-date description is provided here. A formal semantics is not given as the meaning of the language constructs should be fairly clear anyway, and we have already described how to translate programs with procedures into pushdown systems (see Section 2.3), which is more or less precisely what Moped does internally.

To define the syntax formally, we re-use most of the conventions for pushdown systems. We use the following conventions:

- We use the same rules as in pushdown systems for *number*s, whitespace, and the meta-rules for lists; all keywords and other recognized tokens are enclosed in quotes in the syntax description below.
- An *identifier* here is either any alphanumeric string (underscore counting as a letter) other than a keyword, or a sequence of characters other than space, newline, or }, enclosed by curly braces {,}. Case is significant.
- Comments begin with `//` and extend to the end of the line.

Essentially, a Boolean Program consists of a set of procedures. One procedure must be named `main` and it is in this procedure that execution begins. Global variables can be declared at the top of the program. Their values are undetermined at the beginning of the program, i.e. they can be either true or false.

bp : :

decl–list function–list

Since all variables in the program are of type boolean, a declaration is simply a list of variable names:

decl ::
    `decl` *identifier–clist* `";"`

The name of a function is given by an *identifier*; no two functions may have the same name. A function may take zero or more parameters and may return zero or more boolean values (this is defined by the *functype*). The local variable space of a function consists of the parameters and other variables declared inside *funcbody* (see next paragraph).

function ::
    functype identifier funcparams funcbody

functype ::
    `void`
  | `bool`
  | `bool` `"<"` *number* `">"`

funcparams ::
    `"(" ")"`
  | `"("` *identifier–clist* `")"`

The body of a function starts with declarations of its local variables (apart from the parameters). However, additional local variables are implicitly declared 'on the fly' when they occur in the body of the function. A local variable may 'shadow' a global variable. The declarations can be followed by a so-called *enforce* clause. If present, the (boolean-valued) expression inside the enforce clause is a constraint on the global and local variables. Every statement within the procedure is allowed to execute only if the constraint is satisfied. Initially, the local variables can take on any values permitted by the enforce clause.

funcbody ::
    `begin` *decl–list enforce statement–list* `end`

enforce ::
    ε
  | `enforce` *expr* `";"` ;

Expressions are made up of the boolean constants true and false (denoted `T`, `F`, or `1`, `0`, respectively), from identifiers and logical connectives. An identifier must be a global variable or a local variable of the function in which the expression appears. Parentheses are for grouping expressions. The logical connectives are listed in order of precedence, from highest to lowest; they represent negation, equality, inequality, conjunction, exclusive-or (equivalent to inequality), disjunction, and implication. Alternative notations for some operators are $\sim$ for negation, `&&` for conjunction, and `||` for disjunction.

expr ::
  `T | F | 1 | 0`
| *identifier*
| `"("` *expr* `")"`
| `"!"` *expr*
| *expr* `"="` *expr*
| *expr* `"!="` *expr*
| *expr* `"&"` *expr*
| *expr* `"^"` *expr*
| *expr* `"|"` *expr*
| *expr* `"=>"` *expr*

Every statement can optionally be preceded by a 'label' which allows the statement to be addressed as the target of a *goto* statement. A label must be unique within a function. The individual types of statements are discussed below.

statement ::
  optlabel stmt

optlabel ::
  ε
| *identifier* `":"`

stmt ::
  assign
| *call*
| *skip*
| *print*
| *ifstmt*

| *whilestmt*
| *assertion*
| *goto*
| *return*

The first type of statement is parallel assignment. In the grammar rule below, both lists must have the same number of items. The identifiers may be global variables or local variables of the current function. The identifier which occurs in i-th position on the left-hand side will be assigned to the i-th expression on the right-hand side.

assign ::
    identifier–clist `":="` *choose-expr–clist* `";"`

A *choose-expr* is either an *expr* or an `schoose` construct. A term of the form `schoose[`e_1, e_2`]` yields true if e_1 is true, false if e_1 is false and e_2 is true, or is undetermined otherwise.[1]

choose-expr ::
    expr
  | `schoose` `"["` *expr* `","` *expr* `"]"`

The next type of statement is procedure call. If the callee function is of type `void`, i.e. it does not return any values, then the call statement must be of the first form in the grammar rule below. If the callee returns n values, $n \geq 0$, then the statement must be of the second form, and the number of identifiers (global or local variables) on the left-hand side must match the number of return values. In both cases, the number of expressions listed as arguments must match the number of parameters that the function takes. The *identifier* denotes the name of the callee; forward references are allowed.

call ::
    identifier arguments `";"`
  | *identifier–clist* `":="` *identifier arguments* `";"`

arguments ::

[1]The motivation for `schoose` is that c2bp abstracts C code into Boolean Programs in which the variables express predicates of the C program. c2bp generates a statement of the form `p:=schoose[`e_1, e_2`]` if e_1 is a sufficient criterion for predicate p to be true and e_2 is a sufficient criterion for p to be false. Sometimes, however, there is not enough information to determine p, in which case it is allowed to go either way.

```
        "(" ")"
      | "(" choose-expr–clist ")"
```

skip is an 'empty' statement which does nothing. The *print* statement is ignored by Moped and treated just like *skip*.

skip ::
```
        skip ";"
```

print ::
```
        print "(" choose-expr–clist ")" ";"
```

The *ifstmt* has the usual meaning. If the *decider* is satisfied, execution branches into the part after `then`. Otherwise the *decider* expressions in the *elseif* parts are tested, and if all fails, then the *elsepart* is executed (if present).

ifstmt ::
```
        if "(" decider ")" then statement–list elseif–list elsepart fi
```

elseif ::
```
        elsif "(" decider ")" then statement–list
```

elsepart ::
```
        ε
      | else statement–list
```

The truth value of a *decider* is either given by an expression, or is non-deterministic (`*`,`?`).

decider ::
```
        "*" | "?"
      | expr
```

A *whilestmt* iterates over its *statement–list* as long as its *decider* is true.

whilestmt ::
```
        while "(" decider ")" do statement–list od
```

Besides *enforce*, there are three more statements to ensure 'consistency conditions' on the variables: `assume` allows control to continue if the *decider* is true; other paths will be silently aborted. The `assert` statement has the same meaning (in Bebop it also prints an error message when the decider is false, but not in Moped). The `constrain` statement includes a 'freeform' expression which involves the pre- and post-values of the variables; control is allowed to pass the statement and change the variables in such a fashion that their valuations before and after the statement satisfy the expression (very much in the same spirit as the expressions in input language for pushdown systems). In addition to the usual syntax, an *expr* inside the *decider* of a `constraint` statement may contain subexpressions of the form `"'"` *identifier* to refer to the post-values of variables.

> *assertion* ::
>   `assume "("` *decider* `")" ";"`
> | `assert "("` *decider* `")" ";"`
> | `constrain "("` *decider* `")" ";"`

A *goto* statement transfers control to the statement labelled with the given *identifier*. A *goto* statement cannot cross function boundaries.

> *goto* ::
>   `goto` *identifier* `";"`

A *return* statement terminates the execution of a function and returns control back to its caller. If the function returns values, then the second form of the rule below has to be used, and the number of items in the list of expressions must match the number of return values.

> *return* ::
>   `return ";"`
> | `return` *choose-expr–clist* `";"`

A.3 Specifying properties

Moped can perform both reachability analysis and LTL model-checking. In this section we show how to specify properties in Moped. The usage has been demonstrated on some examples in Section 4.1.

Reachability properties When the input to Moped is a pushdown system as described in Section A.1, the form of a reachability formula is

> *pds–reachform* ::
>   *ctrlid* `":"` *stackid*

where the *ctrlid* and *stackid* are a pair $(p, \gamma) \in P_0 \times \Gamma_0$. A reachability query for (p, γ) decides the property

$$post^*(C_0) \cap \{ \langle (p, g), (\gamma, l)\, w \rangle \mid g \in G,\ l \in L,\ w \in (\Gamma_0 \times L)^* \} \neq \emptyset$$

In other words, is there an initial configuration from which a configuration is reachable in which p is the control location and γ is on top of the stack, with arbitrary variable contents.

There is no explicit support in Moped to check for reachability of different categories of configurations. While in principle it would be easy to add support for different sorts of reachability queries, the ability to check for reachability of a 'head' $\langle p, \gamma \rangle$ is the most relevant for many practical problems. But even in the current state of the tool, one could for instance emulate regular languages as reachability targets by non-deterministically allowing the pushdown system to enter the simulation of a finite automaton which recognizes the languages by destructively reading the stack.

For reachability checks on Boolean Programs, the user is expected to pass the name of a label. Moped then checks if there is any execution that leads to the statement with the given label. The label may be given in the form *function*:*label* or, omitting the function name, *label*. The first form resolves ambiguities if the given label occurs in multiple functions; if the second form is used, Moped finds a function in which the label is defined. (Notice that Moped will pick the first function it finds which contains the label, the user should make sure to use this syntax only if the label is unambiguous.)

LTL model-checking Moped relies on Spin [25] to translate LTL formulas into Büchi automata; thus, the format of LTL formulas is taken from there:

ltlform ::
`"true"` | `"false"`
| *proposition*
| *ltlform* `"&&"` *ltlform*
| *ltlform* `"||"` *ltlform*
| *ltlform* `"->"` *ltlform*
| *ltlform* `"<->"` *ltlform*
| `"!"` *ltlform*
| `"[]"` *ltlform*
| `"<>"` *ltlform*
| `"X"` *ltlform*

| *ltlform* "U" *ltlform*
| *ltlform* "V" *ltlform*
| "(" *ltlform* ")"

Basic operands are "true", "false", and atomic propositions. Atomic propositions are slightly different for the two input languages, see below. The logical connectives are, listed above in this order, logical and, logical or, implication, equivalence, and negation. The temporal operators are $\Box$ (rendered as "[]"), $\Diamond$ (rendered as "<>"), the usual $\mathcal{X}$ and $\mathcal{U}$ operators and $\mathcal{V}$, the dual of $\mathcal{U}$.

In the context of pushdown systems the allowed atomic propositions are the identifiers which occur in place of any *ctrlid* or *stackid* in the system. If `r` is such an identifier, then the atomic proposition `r` is true of the configurations

$$\{\, \langle p, (\mathtt{r}, l)w \rangle \mid p \in (P_0 \times G),\ l \in L, w \in (\Gamma_0 \times L)^* \,\} \\ \cup \{\, \langle (\mathtt{r}, g), w \rangle \mid g \in G,\ w \in (\Gamma_0 \times L)^+ \,\}$$

When model-checking LTL on Boolean Programs, the syntax for LTL formulas is basically the same as for pushdown systems; only the atomic propositions are different. Atomic propositions have the same format as in reachability queries, i.e. *function*:*label* or simply *label*; an atomic proposition is true of all program states in which execution has arrived at the statement with the given label.

A.4 Usage and options summary

This section provides a brief overview of Moped's usage and its options. Some examples of the usage have been discussed in Section 4.1. The effect of the options will only be discussed briefly here, but pointers to more thorough treatments are given. In general, the usage is as follows:

`$ moped` $\langle$*modelfile*$\rangle$ $\langle$*formula*$\rangle$

Additional arguments starting with a minus sign are treated as options. Options can be combined, e.g. `-b` and `-r` can be abbreviated as `-br`.

The $\langle$*modelfile*$\rangle$ should be either a pushdown system or a Boolean Program. The latter should be indicated by the `-b` option. The $\langle$*formula*$\rangle$ is either an LTL formula or a reachability formula. Reachability queries are indicated with the `-r` option; see Section A.3 for the syntax of formulas. The available options can be grouped into five categories:

- options defining the task that Moped should solve;
- options influencing the methods which are applied;
- options concerning the use of BDD variables;
- special options for Boolean Programs;
- options which set Moped's level of verbosity.

Below, the options are listed by category.

Setting the task

`-b` As already mentioned above, the user must supply this option to indicate that a Boolean Program is checked; otherwise a symbolic pushdown system is assumed. The syntax of these languages has been described in Sections A.1 and A.2. Supplying `-b` on the command line also selects `-m2v2` (see below) unless overridden by subsequent options.

`-r` The user must supply this to indicate that a reachability analysis should be performed; the default is LTL model-checking. This influences the way the given formula is interpreted, see Section A.3. Note: In the terminology of Section 2.5, the reachability analysis corresponds to problem (I), and LTL model-checking to problem (III).

`-F` In LTL model-checking, this indicates that ⟨*formula*⟩ should not be taken as the LTL formula, but rather as the name of a file which already contains the Büchi automaton for the negation of a formula. The Büchi automaton must be in the format generated by Spin's `-f` command.

`-t` Indicates that a witness path should be generated if a reachability analysis is successful, or a counterexample if LTL model-checking comes to a negative result. For reachability analyses on Boolean Programs the witness path can be printed in different formats. The format is determined by adding another letter:

  `-tP` Default, same as just `-t`. Prints the witness as a path in the pushdown system which Moped generates from the Boolean Program it receives as input.

- `-tN` Outputs the witness path in a format which can be analyzed by Newton (see Section 4.2.3).
- `-tX` Outputs an XML trace. Rajamani has written a viewer which can read these traces and display them in a graphical user interface.

`-D` When checking pushdown systems, this option defines a symbolic constant which can be referenced in the input file, e.g. `-DN=3` would define the symbolic constant `N` to be 3. Such a definition overrides any other definition of `N` in the input file, see also the discussion of *definition* in Section A.1.

Selecting analysis methods

`-p` Reachability of a set of configurations C from the initial configuration c_0 can be expressed as $c_0 \in pre^*(C)$ or $post^*(\{c_0\}) \cap C \neq \emptyset$.[2] We shall call the first the *pre** method and the second the *post** method. We have also explained so-called *pre** and *post** methods for checking LTL, see Section 3.2. Using the `-p` option followed by a digit, the user can specify which type of method should be used:

- `-p0` This instructs Moped to use the *pre** method.
- `-p1` This instructs Moped to use the *post** method.
- `-p2` This instructs Moped to use the *post** method and stop when the first witness/counterexample is found. This method is the default.

The *pre** method has better asymptotic complexity, however in practice the *post** method often performs better (see the discussion in Section 5.5).

`-q` Determines the queueing strategy. When computing forward and backward reachable sets of configurations, Moped picks the transition which it processes from a worklist. The worklist can be organized in different fashions:

- `-q0` Organizes the worklist as a stack, which effectively leads to depth-first search. This is the default.

[2] The classes of sets C which can be specified in Moped is explained in Section A.3.

-q1 Organizes the worklist as a LIFO queue, which means breadth-first search.

These options and their effects are explained in more detail in Section 5.4.

-c Determines the method used for detecting accepting cycles when model-checking LTL. The option is to be followed by one or more capital letters which control two parameters: whether or not to use a preprocessing step, and the method used to detect the cycles. More details on the meaning of these options is given in Section 5.2.

-cT Preprocess by removing trivially irreversible transitions (default).

-cN Do not remove trivially irreversible transitions.

-cC Compute cycles with transitive closure method.

-cE Compute cycles with Emerson-Lei method.

-cO Compute cycles with OWCTY method (default).

-cX Compute cycles with Xie-Beerel method.

Multiple letters can be combined as in, e.g., -cEN.

Setting BDD options

-v In both input languages, we may have a number of different sets of local variables. Therefore, different strategies for mapping the local variables onto BDD variables are possible:

-v0 "avaricious": Determine the number of bits necessary to represent the largest local variable set and use only that many BDD variables (i.e. no more BDD variables than absolutely necessary). This is the default for symbolic pushdown systems.

-v1 "generous": Allocate a fresh set of BDD variables for each set of local variables.

-v2 "by name": This option is available only for Boolean Programs and is the default for them. In this mode, if two local variables in different sets have the same name, they are represented by the same BDD variables.

The effects of these modes are discussed in some more detail in Section 5.9.

`-m` Determines the variable ordering. The following options are available:

`-m0` This is the default when checking symbolic pushdown systems. BDD variables are ordered according to four rules, in the order listed below. 'Input order' refers to the order in which the variables appear in the declarations.

1. scalar variables come before array variables;
2. more significant bits of integers come before less significant bits (boolean variables have just one bit with the least significance for this purpose);
3. variables with earlier appearance in the input order come before those with later appearances;
4. for globals, BDD variables from G_0 come first, then G_1, and then G_2; for locals, L_0 comes first, then L_2, L_1, and L_3.

`-m1` This is an alternative option for symbolic pushdown systems; it is almost like `-m0`, but rules 2 and 3 are swapped.

`-m2` This becomes the default when Boolean Programs are checked. The variable ordering is determined by dependency analysis between the program variables in the input, see Section 5.8.

`-a` This enables the automatic variable reordering feature of CUDD (default is to disable automatic reordering). The reordering operations tend to take very long and thus have a hope of improving overall performance only if Moped's initial variable ordering turns out to be bad. In the experiments reported in this thesis, the initial orderings always turned out to be good enough so that the checker's performance was better with automatic reordering disabled. For experimental purposes, the user may also specify `-aa`, which turns on automatic reordering only while the input is parsed so that the BDD package may make some initial adjustments to the ordering based on the BDDs which represents the pushdown system.

Special options for Boolean Programs

-o The performance of Moped on Boolean Programs can be improved by making use of techniques from compiler optimization, for instance by eliminating dead variables. This is explained in greater detail in Section 5.7.

- -o0 Do not use any optimizations (default).
- -o1 Perform a mod/ref analysis on the global variables.
- -o2 Perform a full 'live range' analysis for all local variables.
- -o3 Use both optimizations.

If no digit is given after -o, then -o3 is assumed. Note: At the time of writing, these optimizations will not work together with -tN and -tX. However, this is not a fundamental incompatibility, again see Section 5.7.

Verbosity

-s Sets the level of Moped's verbosity. The option is to be followed by a single digit which sets the level:

- -s0 Silent mode; Moped merely outputs a yes/no answer and a trace, if applicable.
- -s1 Additionally prints progress and timing reports (default).
- -s2 Additionally prints statistical information from the CUDD package about BDD usage at the end.
- -s3 (and higher) These levels are only for debugging purposes.

Bibliography

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. CAV'01*, LNCS 2102, pages 207–220. Springer, 2001.

[2] H. R. Andersen. An introduction to binary decision diagrams. Technical report, IT University of Copenhagen, Apr. 1998. Lecture notes.

[3] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[4] T. Ball and S. Rajamani. Boolean programs. a model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[5] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[6] M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming*, pages 652–666, 2001.

[7] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model-checking of linear time logic properties. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 222–235. Springer, 1999.

[8] B. Boigelot, J.-M. François, and L. Latour. The Liège Automata-based Symbolic Handler (LASH), 2001. Beta version available from http://www.montefiore.ulg.ac.be/∼boigelot/research/lash/.

[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR'97*, LNCS 1243, pages 135–150, 1997.

[10] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

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

[12] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *Proc. ICALP'97*, volume 1256 of *LNCS*, pages 419–429. Springer, 1997.

[13] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *LNCS*, pages 52–72. Springer, may 1981.

[14] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[15] E. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, B, 1991.

[16] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[17] E. A. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, 1986.

[18] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV 2000*, LNCS 1855, pages 232–247. Springer, 2000.

[19] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proceedings of FoSSaCS'99*, volume 1578 of *LNCS*, pages 14–30. Springer, 1999.

[20] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proceedings of TACS'01*, volume 2215 of *LNCS*, pages 306–339. Springer, 2001.

[21] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural par allel flow graphs. In *Proceedings of POPL '00*, 2000.

[22] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[23] K. Fisler, R. Fraer, G. Kamhi, Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of TACAS'01*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[24] J. Henckel. An efficient linear unification algorithm, 1998. Published at http://www.geocities.com/Paris/6502/unif.html.

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

[26] T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[27] S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Computer Security Foundations Workshop*, 2002. To appear.

[28] O. Kupferman and M. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[29] O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proceedings of CAV 2000*, LNCS 1855, pages 36–52. Springer, 2000.

[30] D. Long. CMU BDD package, 1993. http://emc.cmu.edu/pub.

[31] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU-München, 1998.

[32] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[33] J. Obdržálek. Formal Verification of Sequential Systems with Infinitely Many States. Master's thesis, FI MU Brno, 2001.

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

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

[36] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Proceedings of FM-CAD'00*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.

[37] S. Schwoon. Moped – A Model-Checker for Pushdown Systems, 2002. http://www7.in.tum.de/∼schwoon/moped.

[38] F. Somenzi. Colorado University Decision Diagram Package, 1998. ftp://vlsi.coloradu.edu/pub.

[39] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *Proceedings of CONCUR'95*, volume 962 of *LNCS*, pages 72–87. Springer, 1995.

[40] R. E. Tarjan. Depth first search and linear graph algorithms. In *SICOMP 1*, pages 146–160, 1972.

[41] M. Y. Vardi and P. Wolper. Automata Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[42] I. Walukiewicz. Pushdown processes: Games and model checking. In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 62–74. Springer, 1996.

[43] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of ICCAD*, pages 37–40, San Jose, CA, 1999.