IT Licentiate theses
2000-008

Regular Model Checking

MARCUS NILSSON

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

Regular Model Checking

BY
MARCUS NILSSON

December 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosofy in Computer Systems
at Uppsala University 2000

Regular Model Checking

Marcus Nilsson
marcusn@it.uu.se

Department of Computer Systems
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

We present *regular model checking*, a framework for algorithmic verification of infinite-state systems with, e.g., queues, stacks, integers, or a parameterized linear topology. States are represented by strings over a finite alphabet and the transition relation by a regular length-preserving relation on strings. Both sets of states and the transition relation are represented by regular sets. Major problems in the verification of parameterized and infinite-state systems are to compute the set of states that are reachable from some set of initial states, and to compute the transitive closure of the transition relation. We present an automata-theoretic construction for computing a non-finite composition of regular relations, e.g., the transitive closure of a relation. The method is incomplete in general, but we give sufficient conditions under which it works. We show how to reduce model checking of ω-regular properties of parameterized systems into a non-finite composition of regular relations. We also report on an implementation of regular model checking, based on a new package for non-deterministic finite automata.

Publications

This thesis is based on work, parts of which have previously been published. This is a list of the relevant papers, and notes on my participation in them.

[A] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[B] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. TACAS '00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 2000.

[C] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, *Lecture Notes in Computer Science*, 2000.

Paper [A] is a description of the first model for parameterized systems. My participation was the observation that transitive closures of some types of transition relations could be represented by a finite-state automaton, and showed that the protocols could be verified using this technique. The implementation was also due to me.

Paper [B] is a generalization of [A], introducing the notion of *local depth*. My participation was the main theorem and the implemenation.

Paper [C] was written during my visit in Paris, and is a general description of regular model checking. This paper is more practical than [B] and describes the column transducer construction given in Chapter 6 of this thesis, along with some other techniques based on *widening*. Also, the results on liveness appeared in this paper. My participation in this paper was the column transducer construction and the liveness result, along with writing and implemenation.

Contents

Chapter 1

Introduction

There are many tools for increasing the confidence in that a system works as it should. One of these is the use of *formal methods*, methods based on mathematical models for reasoning about systems. Several attempts have been made to *automate* these reasonings, allowing large systems to be analyzed. In particular, the method of *model checking* [CGP99] has been a successful technique in this direction. In model checking, the system to be analyzed is modeled using some framework based on defining a set of states and a transition relation determining how the system may change over time. This model can then be checked against a *specification*, written in some logic specifying desirable properties of the system's dynamic behavior.

Although model checking has been successful in analyzing fully automatically systems of varying size, its applicability has been limited to systems with a *finite*, and usually a small, number of states. To remedy this, researchers have proposed a number of techniques for handling other types of systems containing components which are inherently infinite state, e.g., Boigelot and Wolper [BW94] for systems containing integer variables. Each of these techniques takes advantage of some regularity in the system to be able to analyze it automatically. The main problem is to define a finite representation of infinite sets of states. For example, the equation $y = 2x$ can be used to represent the set of even numbers y in a finite way. There are several such techniques, each using a particular way of representing an infinite number of states. There is a currently a need, however, to combine these techniques into something more general.

This thesis aims to present a *unifying* framework, called *regular model checking*, still being able to analyze a large class of systems *automatically*. The framework is based on formal language theory, using words over a finite alphabet to describe data-types such as integers and queues. Regular sets are used to represent sets of states and transition relations. The ability to automate model checking in this framework thus depends on the ability to find an encoding into words such that the resulting sets are regular. Even so, the framework is able to include several frameworks in the literature as a special case. Thus, *regular model checking is a unifying framework for automated formal verification of infinite-state systems.*

Let us describe the concept of model checking. One common approach for verification is to describe systems using some logic that describes how the system is supposed to behave, representing assumptions about the system. The specification is written in the same logic, and it is checked whether the description implies the specification. This can be done using semiautomatic theorem provers, e.g. PVS [SOR93], which assist in finding the proofs. In this approach, the description defines not only one model, but all models satisfying the description. In model checking, the model is described directly, more concretely, using some framework often based on sets of states and transitions between the states. The model can be checked against the specification, using some algorithm. This is called *model checking*.

The inherent problem with model checking approaches is the *state-space explosion problem*, that the number of states is very large for most models. This problem has been attacked by introducing compact representations of sets of states allowing larger sets of states to be represented. These representations take advantage of some regular structure of the system. Such a non-direct representation is called a *symbolic* representation, and, accordingly, model checking using symbolic representations is called *symbolic model checking*. The term *symbolic model checking* is used by McMillan [BCMD92, McM93] to denote symbolic model checking based on BDDs (Bryant [Bry86]), a compact representation of finite sets, and has been used with notable success in verification of hardware circuits.

The idea of symbolic model checking has also been applied to *infinite-state systems*, where the representation is actually representing infinite sets of states. For example, there are frameworks based on linear constraints suitable for model checking systems with integer variables (e.g. Boigelot and Wolper [BW94]), and frameworks based on special representations of constraints on clock variables suitable for model checking systems with clock variables (Alur [AD94]), and so on. One problem for these approaches is the lack of methods for combining and unifying these techniques for automatic verification of heterogenous systems using variables of varying types. This problem is addressed by this thesis.

Regular model checking is based on formal language theory, using words (strings) over a finite alphabet to describe states. All data-types such as integers and queues are translated into this word representation. The idea of using regular sets to represent this kind of data-types has been used for integers by Wolper and Boigelot [WB00] and for queues by Boigelot and Godefroind [BG96], encoding the queue content as a word of a special form. For example, an integer value n could be represented in the framework of regular model checking by the word $a^n \perp^m$ for some m over the alphabet $\{a, \perp\}$. The reason for the presence of the symbol $\perp$ is that transitions in our framework will be *length-preserving*, i.e., they do not change the length of the words. Thus, in this example the symbol $\perp$ provides for space to increase the integer variable represented by the word. Sets

of states are represented by regular sets, which allows us to reason about words of arbitrary length. Continuing our example, the set of all even integers could be represented by the set denoted by the regular expression $(aa)^*\perp^*$. Care must be taken not to chose encodings resulting in non-regular sets. Suppose that we choose to represent the value of two integer variables x and y using words over the alphabet $\{x, y, \perp\}$ where the word $x^n \cdot y^m \cdot \perp^k$ for some k denotes a state in which the variable x equals n and the variable y equals m. Then, we can not represent the set of states where the value of x equals the value of y, since the set $\{x^n \cdot y^m \cdot \perp^k \ : \ m = n\}$ is not regular. A better idea would instead be to represent the two integer variables by words over the alphabet $2^{\{x,y\}}$, such that the symbol x is in the set at position n iff the integer variable x equals n and similarly for y. Then, the set $\{\{\}^n \cdot \{x, y\} \cdot \{\}^m \ : \ m, n \geq 0\}$ representing states where x equals y is regular.

Transition relations in regular model checking representing how the system can progress over time are based on *regular relations*, relations represented by automata. One example of a transition relation is a relation relating states where an integer variable is incremented by one. If we represent this integer variable using a word $a^n \cdot \perp^m$, where n represents the value of the variable, this relation would relate words where the second word contains one more a than the first. One often wants to analyze the set of reachable states in a system, starting from the set of initial states and repeatedly applying the transition relation obtaining new states until no new states are found. In this example, this process would never terminate since we would get a new state no matter how many times we apply the transition increasing the integer variable by one. What is needed is a method for analyzing the behavior of a transition when applied an unbounded number of times, in this example yielding the set $a^*\perp^*$ if we start from the set of words $\perp^*$. In terms of regular model checking, we need the ability to calculate non-finite compositions of relations, in this example the transitive closure of a relation. An important contribution of this thesis is a method to compute the result of such compositions. The method is incomplete, as an automaton representing the result need not even exist. A termination condition is given, which also can be used as a characterization of relational compositions that are equivalent to a regular relation.

We use a temporal logic (see e.g. Pnueli [Pnu77]) to specify the desired dynamic behaviour of our systems. We use the well-known result described by Vardi and Wolper [VW86] to translate a temporal formula to a Büchi automaton whose dynamic behavior is characterized by the negation of the formula. In addition, we use regular sets and relations to represent the automata. This allows for some properties to be *parameterized* by the position in the word, allowing for example parameterized fairness conditions. It is shown how to reduce the model checking problem, including the handling of *fairness*, into the calculation of a composition of regular relations.

This thesis also reports on an implementation of regular model checking, and discusses the applicability of the method and the practical problems that have to be solved to make the method efficient. Regular model checking relies on the use of automata, and there are packages implemented for automata used in the context of monadic second-order logic, e.g. MONA [HJJ+96] and Mosel [KMMG97]. We have implemented an automata package for non-deterministic automata with a more direct interface than using the monadic second-order logic, suitable for the implementation of regular model checking. As the other packages, it uses BDDs to represent the transition relation of the automata. While in the case of MONA and Mosel only the alphabet part of the transition relation is reprented with BDDs, we use BDDs to represent the states as well, allowing for some interesting techniques for some of the operations on automata.

1.1 RELATED WORK

In recent years, there has been much effort to extend the wide range of theory and methods for verification of finite-state systems to infinite-state systems, allowing for queues, integers, arrays and other data structures with an infinite domain. Various approaches have been proposed. Typically, a representation for sets of states is proposed, with algorithms to perform transformation of this representation corresponding to operations on sets. These representations are chosen to be able to represent some commonly used domains in systems, such as integers and queues. There is, however, still a lack of methods for combining all these techniques for systems with a variety of domains combined in one system.

Regular model checking uses ideas from formal language and automata theory to obtain a uniform way to represent these different types of systems. If this is the most efficient way to represent systems is not known, but there is evidence, supported by this thesis, that it can be used to uniformly represent a variety of data structures and still be used for automated verification.

Several researchers, e.g., Boigelot and Wolper [WB98] and Kesten et al. [KMM+97], have argued for the advantages of using regular sets as a basis for verifying infinite-state systems. Other researchers, e.g., Fribourg and Olsén [FO97] and Sistla [Sis97], use regular sets in a deductive framework, where basic manipulations on regular sets are performed automatically. These methods are based on proving an invariant given by the user or by some invariant generation technique, but are not fully automatic.

The problem of calculating the effect of arbitrarily long sequences of transitions has been addressed for certain classes of systems, e.g., systems with unbounded FIFO channels [BG96, BGWW97, BH97, ABJ98], systems with pushdown stacks [BEM97, Cau92, FWW97], systems with counters [BW94, CJ98], and certain classes of parameterized systems [ABJN99, CGJ95].

1.2 ORGANIZATION OF THESIS

This thesis is organized as follows. In the next chapter we describe how to model systems and how we specify properties that we would like the system to have, and present examples that we are considering in our framework. Chapter 3 contains the necessary definitions from formal language theory and a discussion of regular relations and their limits. In Chapter 4 we provide a discussion of how to encode different types of infinite-state systems into a model based on regular sets and relations. Model checking using our framework is discussed in Chapter 5, using a toolbox of techniques presented in Chapter 6, dealing with non-finite compositions of regular relations. An implementation of regular model checking is described in Chapter 7, and finally concluding remarks are given in Chapter 8.

Chapter 2

Models

By a model we will refer to a representation of a dynamic behavior using a mathematical description based on sets of states and a transition relation. When modeling systems, we need a representation of the states in the real world. We will call this representation *configurations*. A configuration may for example be the values of some variables in a program or a representation of the content of a network. We may choose different sets of configurations for the same system, representing different views of the same system. The reasons for this may be that we want to analyze different aspects of the systems, or that we want to analyze the system at different levels of abstraction.

When analyzing systems, we look at their *temporal behavior*. The temporal behavior of a system is how the system state changes with time. We will adopt the view of looking at sequences of states, or configurations in our representation, where positions in the sequence represents the time line and the content at each position represents the configuration at that particular moment.

We introduce the basic notions for temporal behavior. Let Γ be a set of *configurations*. A *run* θ over Γ is an infinite sequence of configurations from Γ. We use Γ^ω to denote the set of all runs over Γ.

Example 2.1 Consider a system consisting of a counter. The counter begins at zero and increments its value by one in each time step. To model this system, we choose as the set of configurations Γ the set of natural numbers $\mathcal{N}$. A run in this system would start at zero and increase by one at each time step, represented by the run $0\ 1\ 2 \cdots \in \Gamma^\omega$. □

We will describe two different ways of describing systems with a particular behavior. One is the use of a *model*, which is similar to a state machine and can be used to model directly the systems we want to analyze. The other is the use of a *temporal logic*, a logic for reasoning about the behavior itself. This logic will be used for *specification*.

2.1 MODEL

A widely used model for describing temporal behavior is a Büchi automaton, introduced by Büchi.

Definition 2.2 Let Γ be a set of *configurations*. A *Büchi automaton model* $\mathcal{M}$ over Γ is a tuple $(\Gamma^I, \longrightarrow, \Gamma^F)$ where

- Γ^I is a subset of Γ, called the set of *initial configurations*, and
- $\longrightarrow$ is a relation on $\Gamma \times \Gamma$, called the *transition relation*, and
- Γ^F is a subset of Γ, called the set of *accepting configurations*.

□

We will use the term *model* to mean a Büchi automaton model. The set of initial configurations represents the set of initial states in the system. The transition relation represents the behavior of the system in one step. If a pair of configurations is in the relation, it means that the system can make a step from the first configuration to the second. Following the transition relation starting from the initial configuration we get a *run* of this system, defined below. The set of accepting states is used to specify additional constraints on the runs.

Let $\mathcal{M} = (\Gamma^I, \longrightarrow, \Gamma^F)$ be a model over Γ. A *run* of $\mathcal{M}$ is a run $\gamma_0\gamma_1\cdots$ over Γ such that $\gamma_i \longrightarrow \gamma_{i+1}$ holds for all i with $i \geq 0$. An *accepting run* of $\mathcal{M}$ is a run $\gamma_0\gamma_1\cdots$ of $\mathcal{M}$ such that there is a configuration $\gamma \in \Gamma^F$ and an infinite set of indices I such that $\gamma_i = \gamma$ for all $i \in I$. We use $[\![\mathcal{M}]\!]$ to denote the set of all accepting runs of $\mathcal{M}$. When $\Gamma^F = \Gamma$, the model behaves like an ordinary state machine.

2.2 DESCRIBING SYSTEMS

To describe systems, we use the model from the previous section where the set of accepting states is the set of configurations. This way, all runs of the model are accepting runs.

We usually choose a tuple of variables $V = (x_1, x_2, \ldots, x_n)$ together with their domains $\mathcal{D} = (D_1, D_2, \ldots, D_n)$. The set of configurations Γ is then $D_1 \times D_2 \times \cdots \times D_n$, the cross product of the domains. We use predicates over variables to describe sets of configurations, for example $x_3 = 5$ describes the set $D_1 \times D_2 \times \{5\} \times D_4 \times \cdots \times D_{n-1} \times D_n$. To describe relations on $\Gamma \times \Gamma$, we use an unprimed version of the variables to represent the first component and a primed version of the variables to represent the second component. For example, $x_3 = x'_3$ represents the relation between configurations where the value of x_3 is the same.

Example 2.3 Consider a token ring system consisting of n processes connected in a ring shaped network. We represent this system using a variable q ranging over $\{N, T\}^n$, the set of words of length n over the alphabet $\{N, T\}$, where N represents a process which does not have the token and T represents a process which has the token. For i with $1 \leq i \leq n$, we use $q[i]$ to denote the content of the word q at position i. Thus, $q[i]$ represents the state of process i, where the processes are ordered from 0 and upwards in the order they appear in the ring. The set of configurations is then $\Gamma = \{N, T\}^n$.

The set of initial states Γ^I is given by the set of words $N^i T N^j$ such that $i+j+1 = n$. The transition relation $\longrightarrow$ is given by the relation where one process sends the token to its neighbor which can then be represented by

$$\exists i : \begin{array}{c} \forall j < i : q'[j] = q[j] \\ \wedge \\ q[i] = T \ \wedge \ q'[i] = N \\ \wedge \\ q[i+1] = N \ \wedge \ q'[i+1] = T \\ \wedge \\ \forall j > i+1 : q'[j] = q[j] \end{array}$$

where all arithmetic is done modulo n. The set of accepting states Γ^F is set to Γ. □

2.3 SPECIFICATION USING TEMPORAL LOGIC

To specify behavior, we use a *temporal logic*. A temporal logic specify behavior directly rather than a description of a system having a particular behavior. Thus, using such a logic, we can more easily specify how we want the system to behave without thinking about a particular system.

There are many different types of temporal logics. We present a version called the *propositional temporal logic* and state a well-known result to translate this logic into a model having the same behavior as a formula in this logic.

As the atomic propositions, we take predicates over the set of configurations, or equivalently, subsets of configurations. The logic will thus be parameterized by the set of configurations, and we will use $PTL(\Gamma)$ to denote the propositional temporal logic using subsets of Γ as atomic propositions. More formally, the logic is defined as follows.

Definition 2.4 Let Γ be a set of configurations. The *propositional temporal logic* over Γ, denoted $PTL(\Gamma)$, is defined as the least set closed under the following rules:

1. $2^\Gamma \subseteq PTL(\Gamma)$

2. If $\varphi_1, \varphi_2 \in PTL(\Gamma)$, then $\varphi_1 \ \vee \ \varphi_2 \in PTL(\Gamma)$.
3. If $\varphi \in PTL(\Gamma)$, then $\neg\varphi \in PTL(\Gamma)$.
4. If $\varphi \in PTL(\Gamma)$, then $\Box\varphi \in PTL(\Gamma)$.
5. If $\varphi \in PTL(\Gamma)$, then $\circ\varphi \in PTL(\Gamma)$.
6. If $\varphi_1, \varphi_2 \in PTL(\Gamma)$, then $\varphi_1 \mathcal{U} \varphi_2 \in PTL(\Gamma)$.

□

Intuitively, the formula $\Box\varphi$ states that φ holds now and forever at all points in the future, and the formula $\circ\varphi$ that φ holds at the next point of the time line, and the formula $\varphi_1 \mathcal{U} \varphi_2$ that φ_1 holds until φ_2 holds.

For a set Γ of configurations, each formula $\varphi \in PTL(\Gamma)$ denotes a set $[\![\varphi]\!]$ of runs over Γ, defined by the following rules:

1. $[\![\Gamma_0]\!]$ is the set of runs $\gamma_0\gamma_1 \cdots \in \Gamma^\omega$ such that $\gamma_0 \in \Gamma_0$, for all $\Gamma_0 \subseteq \Gamma$
2. $[\![\varphi_1 \ \vee \ \varphi_2]\!] = [\![\varphi_1]\!] \ \cup \ [\![\varphi_2]\!]$.
3. $[\![\neg\varphi]\!] = \Gamma^\omega \setminus [\![\varphi]\!]$.
4. $[\![\Box\varphi]\!]$ is the set of runs $\gamma_0\gamma_1 \cdots \in \Gamma^\omega$ such that the run $\gamma_i\gamma_{i+1} \cdots$ is in $[\![\varphi]\!]$ for all $i \geq 0$.
5. $[\![\circ\varphi]\!]$ is the set of runs $\gamma_0\gamma_1 \cdots \in \Gamma^\omega$ such that the run $\gamma_1\gamma_2 \cdots$ is in $[\![\varphi]\!]$.
6. $[\![\varphi_1 \mathcal{U} \varphi_2]\!]$ is the set of runs $\gamma_0\gamma_1 \cdots \in \Gamma^\omega$ such that there is an $i \geq 0$ such that $\gamma_i\gamma_{i+1} \cdots$ is in $[\![\varphi_2]\!]$ and $\gamma_j\gamma_{j+1} \cdots$ is in $[\![\varphi_1]\!]$ for all $j < i$.

We introduce the usual abbreviations for $\wedge$, $\Longleftrightarrow$, and $\Longrightarrow$. Also, we introduce the eventuality operator $\Diamond$. The formula $\Diamond\varphi$ is an abbreviation for $\neg\Box\neg\varphi$, and means at some point in the future, φ will hold.

There is a classical result saying that for every formula there is a model with the same runs as the formula (see for example Büchi [Buc62] and Vardi and Wolper [VW86]). The model simulates the behavior of the formula by observing the configurations and, using a *finite* set of internal states, has exactly the runs that is described by the formula. The model will be over configurations both from the configurations of the formula and from the internal state. To formulate this result, we need projections allowing us to talk about these two components. Let Γ, Γ' be sets of configurations. A *projection* π from Γ to Γ' is a mapping from Γ to Γ'. We extend projections to runs by defining $\pi(\gamma_0\gamma_1 \cdots) = \pi(\gamma_0)\pi(\gamma_1) \cdots$.

Theorem 2.5 *Let Γ be a set of configurations. For every formula $\varphi \in PTL(\Gamma)$, there is a finite set of configurations Γ' and a model $\mathcal{M}$ over $\Gamma \times \Gamma'$ such that $\pi([\![\mathcal{M}]\!]) = [\![\varphi]\!]$, where π is the projection $\pi(\gamma, \gamma') = \gamma$.*

Proof: See for example [VW86]. □

2.4 MODELING INFINITE-STATE SYSTEMS

In this section we show several examples of how to model infinite-systems. Common to all these examples are that they are amenable to encoding of their state into words in a way such that the many sets of words that we want to use during verification as a representation of sets of states are regular. In some cases the encoding makes some transitions *atomic* in the sense that conditions that would otherwise translate into loops that checks some conditions becomes a single atomic check. These encodings into finite words are discussed in Chapter 4.

2.4.1 The Bakery Algorithm

In the bakery algorithm for mutual exclusion due to Lamport [Lam74], there are an arbitrary number of processes waiting to get a "ticket" to get into the critical section. Each process which wants to get into the critical section receives a ticket which is the maximum of all the outstanding tickets plus one. When a process has the lowest outstanding ticket, it enters the critical section and drops the ticket when leaving.

To model this algorithm, we use a variable m ranging over a multiset over the set of tuples $\mathcal{N} \times \{T, C\}$ where the first component represents the ticket of a process and the set $\{T, C\}$ represents a control state where T denotes that the process is trying to enter the critical section, and C denotes that the process is in the critical section. Processes that are neither trying to get into or are in the critical section are not represented in the model. We denote by $max(m)$ the maximum value of the tickets of all elements in m, and by $min(m)$ the minimal value of the ticket of all elements in m. The transition relation is then given as follows:

- The case where one process obtains a ticket is given by the relation $m' = m \cup \{(max(m) + 1, T)\}$.
- The case where one process enter the critical section is given by the relation $\exists i \geq 0 : i = min(m) \wedge (i, T) \in m \wedge m' = m \setminus \{(i, T)\} \cup \{(i, C)\}$.
- The case where one process leaves the critical section is given by the relation $\exists i \geq 0 : (i, C) \in m \wedge m' = m \setminus \{(i, C)\}$.

The mutual exclusion property that states that there are never two processes in the critical section is given by the temporal formula $\Box\neg(\Sigma_{i \geq 0} m((i, C))) > 1$.

2.4.2 Szymanski's Algorithm

In the previous example there was an arbitrary number of processes, but there was a complete symmetry between the processes. In this example we will look at another algorithm that works for an arbitrary number of processes, but with the difference that they will be organized in a linear array and thus will not be completely symmetric with respect to each other.

In Szymanski's Algorithm for mutual exclusion[Szy90, GZ98], there are an arbitrary number of processes organized in a linear array, where the index of the array denotes the process ID. In the algorithm, the local state of each process i consists of a control state $pc[i]$, ranging over the integers from 1 to 7 and of two boolean flags, $w[i]$ and $s[i]$. A process i is in the critical section when the control state $pc[i]$ is equal to 7. We model this using three variables ranging over an array of the same length as the number of processes, named pc, and w, and s. The transition relation is given by the following program for each process i, expressed in pseudo-code where the lines are numbered with the value of the control state pc.

1: **await** $\forall j : j \neq i : \neg s[j]$
2: $w[i], s[i] := \mathit{true}, \mathit{true}$
3: **if** $\exists j : j \neq i : (pc[j] \neq 1) \wedge (pc[j] \neq 2)$
  then $s[i] := \mathit{false}$; goto 4
  else $w[i] := \mathit{false}$; goto 5
4: **await** $\exists j : j \neq i : s[j] \wedge \neg w[j]$ **then** $w[i], s[i] := \mathit{false}, \mathit{true}$
5: **await** $\forall j : j \neq i : \neg w[j]$
6: **await** $\forall j : j < i : \neg s[j] \ \vee \ \neg w[j]$
7: $s[i] := \mathit{false}$; goto 1

Figure 2.1: Szymanski's Algorithm

For instance, according to the statement at line 6, if the control state of a process i is 6, and the value of s is *false* in all processes with a lower index, i.e., for all processes j with $j < i$, then the control state of process i may be changed to 7. In a similar manner, according to the statement at line 4, if the control state of a process i is 4, and if there is at least another process j (either with a lower index or a higher index than i) where the value of $s[j]$ is *true* and the value of $w[j]$ is *false*, then the control state, $w[i]$, and $s[i]$, in i may be changed to 5, *false*, and *true*, respectively.

To see how the above statements are modeled, line 1 can for example be modeled by the following transition relation for all i with $1 \leq i \leq n$:

$$pc[i] = 1 \ \wedge \ pc'[i] = 2 \ \wedge \ w'[i] = w[i] \ \wedge \ s'[i] = s[i] \ \wedge \ \forall j : j \neq i : \neg s[j]$$

The mutual exclusion property that states that there are never two processes in the critical section is given by the temporal formula $\Box\neg\exists i, j : i \neq j : (pc[i] = 7 \ \wedge \ pc[j] = 7)$.

2.4.3 Dijkstra's Algorithm

In Fig. 2.2, we show an idealized version of Dijkstra's protocol[LPS93] for ensuring mutual exclusion among an arbitrary number of processes. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1, 2\}$. Furthermore, a global variable p ranging over process indices is used. In the algorithm, line 6 represents the critical section.

```
1:    flag[i] := 1
2:    if p ≠ i then
          await flag[p] = 0 then
3:            p := i
4:    flag[i] := 2
5:    if ∃j ≠ i : flag[j] = 2 then goto 1
6:    flag[i] := 0
7:    goto 1
```

Figure 2.2: Dijkstra's Algorithm

The mutual exclusion property that states that there are never two processes in the critical section is given by the temporal formula $\Box\neg\exists i, j : i \neq j : (pc[i] = 6 \ \wedge \ pc[j] = 6)$.

2.4.4 Burns Algorithm

Burns's Mutual Exclusion Algorithm[LPS93] is given in Fig. 2.3. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1\}$. The critical section is represented by line 6.

```
1:    flag[i] := 0
2:    if ∃j < i : flag[j] = 1 then goto 1
3:    flag[i] := 1
4:    if ∃j < i : flag[j] = 1 then goto 1
5:    await ∀j > i : flag[j] ≠ 1
6:    flag[i] := 0
7:    goto 1
```

Figure 2.3: Burns's Algorithm

The mutual exclusion property that states that there are never two processes in the critical section is given by the temporal formula $\Box\neg\exists i, j : i \neq j : (pc[i] = 6 \ \wedge \ pc[j] = 6)$.

2.4.5 The Alternating Bit Protocol

We consider the well-known Alternating Bit Protocol[BSW69], a protocol used for delivering messages over unbounded channels which are faulty in the sense that they may loose messages but not reorder them.

There are two channels, one for sending messages from the sender to the receiver, and one for sending acknowledgements from the receiver to the sender. Each message is given a sequence number and the sender waits for an acknowledgement from the receiver before sending a new message. Until this acknowledgement is received, the sender may resend the message. When the receiver has acknowledged the message, the procedure is repeated but with the sequence number inverted. Both the sender and the receiver ignore messages with unexpected sequence numbers.

To model the protocol, we consider two operations **send** and **receive**, modeling calls from the upper layers of the protocols. Thus, **send** denotes that there is a new message from the sender side, and **receive** denotes that the receiver side signals that a message has been received. We denote the two channels c_M and c_A , where c_M is the channel used for messages and c_A is the channel used for acknowledgements. We denote by $c!v$ the operation of sending or acknowledging a message with sequence number v to the channel c, and by $c?v$ the operation of receiving a message or acknowledgement of a message with sequence number v from channel c.

The code for the sender and the receiver is given below. The notion $S\mathbf{OR}S'$ means that either S or S' is executed, but not both of them.

Sender		Receiver	
1:	**send**	1:	(c_M?1, c_A!1, **goto** 1) **OR** c_M?0
2:	(c_M!0, c_A?1, **goto** 2) **OR** c_A?0	2:	**receive**
3:	**send**	3:	(c_M?0, c_A!0, **goto** 3) **OR** c_M?1
4:	(c_M!1, c_A?0, **goto** 4) **OR** c_A?1	4:	**receive**
5:	**goto** 1	5:	**goto** 1

One property of the algorithm states that the operations **send** and **receive** alternates after each other such that the two operations never occur consecutively. A temporal formula for this is

$$\mathbf{send} \wedge \Box(\mathbf{send} \implies \neg\mathbf{send}\mathcal{U}\mathbf{receive}) \wedge \Box(\mathbf{receive} \implies \neg\mathbf{receive}\mathcal{U}\mathbf{send})$$

2.4.6 The Sliding Window Protocol

In a sliding window protocol (for a general description on sliding window protocols, see e.g. [Tan96] ch. 3), there are two processes sending messages over a channel. The channel is not perfect, but can lose messages at any time. The goal of the protocol is to receive the messages in order. To accomplish this, a sequence number is assigned to each message that is sent. The numbers are taken from a *sending window* consisting of a range of sequence numbers which defines the set of messages that is currently being sent but which have not yet been acknowledged. As the receiver acknowledges the messages, the sending window is decreased. The sender

may send new messages and thus increasing the sending window, but only up to the *sending window limit*, defining the maximal size of the window.

To model this algorithm, we use three integer variables: low and $high$ defining the current sending window, and $next$ defining the sequence number of the next message to receive. We denote the sending window limit by max, and use a variable c ranging over the set of sequences of integers to model the channel. The integers denote the sequence numbers of the messages in the channel. The acknowledgements are not modeled, but are assumed to happen sychronously between the receiver and the sender.

The transition relation is given by the union of the following transitions, where all operations on the integers are assumed to be modulo max.

- (enlarge window) **if** $low \neq high + 1$ **then** $high := high + 1$
- (send) $\forall n : low \leq n < high : \text{send}(n)$
- (receive) $\text{receive}(next), next := next + 1$
- (synchronous ack) $low := next$

A formula stating that the receiver is never outside the sending window is $\Box low \leq next \leq high$.

2.4.7 A Termination Detection Algorithm

We describe an algorithm for termination detection among an arbitrary number of processes organized in a ring shaped network, found in Dijkstra et. al [DFvG83]. The algorithm uses a colored token which is passed around the ring to check that all processes in the ring have terminated.

A process can either be *non-idle* or *idle*. When all processes are idle, we say that the system has terminated. A process can spontaneously change its state from non-idle to idle, i.e., it terminates. To detect that all processes are idle, a designated processes sends out a token which it colors *white*. When the token is passed to the next processes, the process passing the token paints it black if it is non-idle. When the token comes back to the process which sent out the token, it is white if the system has terminated, and black otherwise.

The system can be modeled by numbering the processes from 1 to n and using three arrays holding three local variables the processes. Only process 1 may initiate the algorithm by sending out a new token. The three variables are $q[i]$ which is true iff process i is idle, $t[i]$ ranging over $\{\textbf{black}, \textbf{white}, \textbf{none}\}$, which has the value **none** when process i does *not* have the token, and otherwise denotes the color of

the token. In addition, process 1 has a boolean variable w, which is true if it has stayed idle during the current round. The value of w is only relevant for process 1.

Initially, we have $q[i] = \mathit{false}$ for all i, and $t_0 = \textbf{black}$, and $t[i] = \textbf{none}$ for all $1 \leq i < N$, and $w = \mathit{false}$. The algorithm can be described by a union of the following transitions, for each process i:

- $q[i] := \mathit{true}$
- **if** $i > 1 \ \wedge \ \neg q[i-1]$ **then** $q[i] := \mathit{false}$
- **if** $\neg q_n$ **then** $q[1], w := \mathit{false}, \mathit{false}$
- **if** $i = 1 \wedge q[1] \wedge (t[1] = \textbf{black} \vee \neg w)$ **then** $t[1], t[2], w := \textbf{none}, \textbf{white}, \mathit{true}$
- **if** $i < n \ \wedge \ t[i] \neq \textbf{none} \ \wedge \ q[i]$ **then** $t[i], t[i+1] := \textbf{none}, t[i]$
- **if** $i < n \ \wedge \ t[i] \neq \textbf{none} \ \wedge \ \neg q[i]$ **then** $t[i], t[i+1] := \textbf{none}, \textbf{black}$
- **if** $i = n \ \wedge \ t[n] \neq \textbf{none} \ \wedge \ \neg q[n]$ **then** $t[n], t[1] := \textbf{none}, \textbf{black}$

The three first types of statements describe the underlying computation: A process can become idle autonomously (first statement), it can become non-idle if its predecessor is non-idle. In addition (third statement), process 1 must set w to *false* if it becomes non-idle. The fourth statement starts a round of the detection algorithm, In the next statement, a process just forwards the token if it is idle. If the process is non-idle, then the token is painted black and then forwarded.

The correctness of the protocol can be stated as $\Box(t[1] = \textbf{white} \wedge w) \implies \forall i : q[i]$, saying that if process 1 signals termination, then all processes are idle.

Chapter 3

Regular Relations

Regular model checking is based on formal languages and automata. In this chapter, we introduce the basic notions of formal language theory and introduce the concept of *regular relations*.

3.1 REGULAR LANGUAGES AND AUTOMATA

We introduce the notion used for regular languages and automata.

Languages Let Σ be a finite set, called the *alphabet*. A *finite word* w over Σ is a finite sequence of symbols from Σ. We use $|w|$ to denote the length of w. We use Σ^* to denote the set of finite words over Σ. A *language* L over Σ is a subset of Σ^*.

Automata A finite automaton A over Σ is a tuple (Q, S, Δ, F) where Q is a finite set of *states*, $S \subseteq Q$ is a finite set of *initial states*, $\Delta : Q \times \Sigma \times Q$ is a *transition relation*, and $F \subseteq Q$ is a finite set of *accepting states*. We lift Δ to words such that $\Delta(q, a_1 a_2 \cdots a_n, q')$ holds iff there are states $q_0, q_1, \ldots, q_n$ with $q = q_0$, and $q' = q_n$, and $\Delta(q_{i-1}, a_i, q_i)$ for all i with $1 \leq i \leq n$. For a set of states $Q_0 \subseteq Q$, the image $\Delta(Q_0, w)$ is defined as the set of states $q' \in Q$ such that $\Delta(q, w, q')$ holds for some state $q \in Q_0$. For a state $q \in Q$, the *set of prefixes* of q, denoted $\mathsf{pref}(q)$, is defined as the set of words w such that $q \in \Delta(S, w)$, and the *set of suffixes* of q, denoted $\mathsf{suff}(q)$, is defined as the set of words w such that $\Delta(q, w) \cap F \neq \emptyset$. For a set of states $Q_0 \subseteq Q$, the set of prefixes $\mathsf{pref}(Q_0)$ is defined as the union of all sets $\mathsf{pref}(q)$ where $q \in Q_0$, and the set of suffixes $\mathsf{suff}(Q_0)$ is defined as the union of all sets $\mathsf{suff}(q)$ where $q \in Q_0$. The language recognized by A, denoted $\mathcal{L}(A)$, is defined as the set $\mathsf{suff}(S)$ of suffixes of the set of initial states. A language L is *regular* iff it is recognized by some automaton.

3.2 REGULAR RELATIONS

Regular model checking is based on *regular relations*, which will be used to represent the transition relations. They are recognized by automata in a similar way as regular sets are recognized by automata. Let us explain this idea more precisely.

Let $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ be finite alphabets. For words $a_1^j \cdot a_2^j \cdots a_n^j \in \Sigma_j^n$ of equal length n for j with $1 \leq j \leq m$, their *cross product*[1]

$$a_1^1 \cdot a_2^1 \cdots a_n^1 \times a_1^2 \cdot a_2^2 \cdots a_n^2 \times \cdots \times a_1^m \cdot a_2^m \cdots a_n^m$$

is defined as the word

$$(a_1^1, a_1^2, \ldots, a_1^m) \cdot (a_2^1, a_2^2, \ldots, a_n^m) \cdots (a_n^1, a_n^2, \ldots, a_n^m).$$

over the alphabet $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m$.

A language consisting of cross products denotes a relation in the following way. For a language L over $\Sigma_1 \times \Sigma_2 \times \cdots \Sigma_m$, we denote by $[L]$ the relation consisting of the set of tuples $(w_1, w_2, \ldots, w_m)$ such that $w_1 \times w_2 \times \cdots \times w_m$ is in L. Note that for $n = 1$, we have that L equals $[L]$.

Relations that can be represented by a regular language in this way are called *regular*.

Definition 3.1 Let $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ be finite alphabets. A relation $R \subseteq \Sigma_1^* \times \Sigma_2^* \times \cdots \times \Sigma_m^*$ of arity m is *regular* if there is a regular language L over $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m$ such that $[L] = R$. □

Compositionality For two regular relations R and R' of equal arity, we define their *concatenation* $R \cdot R'$ as the regular relation $[L \cdot L']$ denoted by the concatenation of the languages L and L such that $R = [L]$ and $R' = [L']$. Their *union* is denoted by $R \cup R'$ and their *intersection* by $R \cap R'$. Regular relations are closed under union and intersection.

Theorem 3.2 *Let R and R' be regular relations of arity k. Then $R \cup R'$ and $R \cap R'$ are regular.*

Proof: We prove that $[L] \cup [L'] = [L \cup L']$ for two languages L and L' such that $R = [L]$ and $R' = [L']$. Let $(w_1, w_2, \ldots, w_k) \in [L \cup L']$. This holds iff $w_1 \times w_2 \times \cdots \times w_k$ is in L or L' which is true iff $(w_1, w_2, \ldots, w_k)$ is in $[L]$ or $[L']$, i.e., $[L] \cup [L']$. The case for intersection can be proved similarly. □

[1] The term "cross product" for finite words is taken from [KMM+97]

For two relations R of arity m and R' of arity m', we define their *length-preserving cross product* $R \overline{\times} R'$ as the set of tuples $(w_1, w_2, \ldots, w_m, w'_1, w'_2, \ldots, w'_{m'})$ such that all words $w_1, w_2, \ldots, w_m, w'_1, w'_2, \ldots, w'_{m'}$ are of the same length and $(w_1, w_2, \ldots, w_m)$ is in R and $(w'_1, w'_2, \ldots, w'_{m'})$ is in R'. For a relation R of arity m, the *projection* $\pi_{(i_1,i_2,\ldots,i_k)}(R)$ on R on a tuple of indices $(i_1, i_2, \ldots, i_k) \in \{1, 2, \ldots, m\}^k$ is defined as the relation of arity m consisting of the set of tuples $(w_{i_1}, w_{i_2}, \ldots, w_{i_k})$ such that there exist a tuple $(w_1, w_2, \ldots, w_m)$ in R. Regular relations are closed under these operations.

Theorem 3.3 *Let R be a regular relation of arity m and let R' be a regular relation of arity m'. Then the following relations are regular*

1. $R \overline{\times} R'$
2. $\pi_{(i_1,i_2,\ldots,i_k)}(R)$, *for all k and* $(i_1, i_2, \ldots, i_k) \in \{1, 2, \ldots, n\}^k$.

Proof: To see (1), consider taking the intersection of the two automata representing R and R', where elements of the two relations are disjoint. It is not hard to see that the resulting automaton will only accept words in the cross product that have the same length.

For (2), apply projection on the transition relation of the automaton (which is finite). □

We will be particularly interested in *binary* regular relations, since they will be used to represent the transition relations in our programs. The *relational composition* of binary regular relations is important because it is used to reason about the progress of time of a system. If R represents a transition relation in one step in a program, then $R \circ R$ represents the transition from one state to another state in *two* steps. Binary regular relations are closed under the relational composition operator $\circ$.

Theorem 3.4 *Let R and R' be binary regular relations on Σ. Then their relational composition $R \circ R'$ is regular.*

Proof: $R \circ R' = \pi_{(1,3)}(R \overline{\times} \Sigma^* \ \cap \ \Sigma^* \overline{\times} R')$ □

For a regular language L, we use Id_L to denote the regular identity relation restricted to L, i.e., the set of pairs (w, w) such that $w \in L$. For a regular relation R and a regular language L, we note that the image of $R(L)$ under L is regular since $R(L)$ is the regular relation $\pi_{(2)}(Id_L \circ R)$.

For a regular relation R, the *transitive closure* of R is denoted by R^+ and the *reflexive and transitive closure* of R is denoted by R^*. If R represents a transition

relation in a program, then R^* represents transitions from one state to another in zero or more steps. Regular relations are not closed under this operation.

Theorem 3.5 *There is a regular relation R such that R^* is not regular.*

Proof: There are many possible counter examples of which perhaps the simplest is that the transition relation of a Turing machine can be encoded as a regular relation. We describe one counter example based on having to match the number of occurences of two symbols. Let $\Sigma = \{a, b\}$ and $R = [((a, a) + (b, b))^*(a, b)(b, a)((a, a) + (b, b))^*] = \{(wabw', wbaw') \ : \ w, w' \in \Sigma\}$ be a regular relation on $\Sigma^* \times \Sigma^*$. If R^* is regular, then the image $R^*(L)$ under the regular language $L = (ab)^*$ is regular. Let $\#c(L')$ denote the number of occurrences of the symbol c in the language L'. For a language L', the relation R preserves the number of a's and b's, i.e., $\#a(L') = \#a(R(L'))$ and $\#b(L') = \#b(R(L'))$. Further, we have $\#a(L) = \#b(L)$. Now consider the language L_i denoting the left quotient of i number of a in the image $R^*(L)$, defined as the set of words w such that $a^i w \in R^*(L)$. We have that $\#b(w) = \#a(w) + n$. It is easy to see that each L_i is non-empty and it follows that each L_i is a different language. Thus, R^* is not regular. □

The above result is not surprising, since regular relations can be used to represent for example the transition relation of a Turing machine. Relational compositions of regular relations will still be used, however, as a basis for our theory. In Section 6, we present a semi algorithm for computing an automaton recognizing a composition built up from $\cup$, $\circ$, and *.

Chapter 4

Regular Models

We have shown how to use a model to describe a variety of different classes of infinite-state systems. To perform automated verification, we will translate these models into a model called *regular model*. This translation from various classes of models into the regular model is what makes regular model checking a unifying framework.

Definition 4.1 Let Σ be an alphabet. A *regular model* over Σ is a model $(\Gamma^I, \longrightarrow, \Gamma^F)$ over Σ^* where Γ^I, $\longrightarrow$, and Γ^F are regular. □

In a regular model, the sets and the relations are regular. Thus, they can be represented by a finite-state automaton which we will use for the algorithms that perform the verification.

To transform a model into a regular model, one chooses a representation of the system state such that each state of the system is represented as a word over some alphabet. The initial set of states is formulated as a regular set over this alphabet, and the transition relation as a regular relation on this alphabet. One must be careful to choose the representation such that the initial set of states is regular. For example, suppose that we want to represent two integer variables x and y using the alphabet $\Sigma = \{x, y\}$, and that we choose to represent a state where $x = n$ and $y = m$ with the word $x^n y^m$. Then we can not represent the set of states where $x = y$, because the set $\{x^n y^n \ : \ n \geq 0\}$ is not regular. If, however, we choose to represent the two integer variables using two boolean variables b_x and b_y yielding the alphabet $\Sigma = \{true, false\} \times \{true, false\}$, the cross product of the domains of b_x and b_y, and to represent a state where $x = n$ and $y = m$ with the word w such that the symbol at position i is in $b_x = true$ iff $n = i$ and in $b_y = true$ if $m = i$, then the set of words representing $x = y$ is regular, namely the set given by the regular expression $(\neg b_x \ \wedge \ \neg b_y)^* \cdot (b_x \ \wedge \ b_y) \cdot (\neg b_x \ \wedge \ \neg b_y)^*$.

In this chapter, we discuss the translation from models to a regular models. The ability to perform automatic verification is largely dependent on how we make this translation.

In Sect. 2.4, we showed several examples of infinite-state systems. We will discuss general principles for deriving regular models from the following types of systems, which occur in the examples:

- **Parameterized Systems** An arbitrary number of homogeneous processes possibly organized in some topology.
- **Integer Variables** Variables ranging over the natural numbers, which is an infinite domain.
- **Queues** Queues between processes, modeling for example communication links.

4.1 PARAMETERIZED SYSTEMS

Consider a system parameterized by the number of processes. Typical examples are algorithms designed to work for an arbitrary number of processes. In this case, we want to verify the system regardless of the number of processes.

We assume that all processes are homogeneous, i.e., all processes have the same set of states Q. Using our representation, we can represent parameterized systems in which the processes are ordered in a linear array. As the alphabet, we take the set of states for each process, i.e., $\Sigma = Q$. Each word $a_1 a_2 \cdots a_n \in \Sigma^*$ is then used to represent a state where process at position i is in state a_i for all i with $1 \leq i \leq n$.

Local transitions not depending on the other processes can be represented by the regular relation $Id_{Q^*} \cdot [(q, q')] \cdot Id_{Q^*}$ where a process can make a transition from q to q'. Other transitions need global conditions, for example that all processes at a position with a lower index should be in a particular state, say q_g. If the processes are ordered in our representation such that a process with index i is represented by the symbol at position i in the word, then we can represent such a transition by the regular relation $Id_{q_g^*} \cdot [(q, q')] \cdot Id_{Q^*}$, where a process can make a transition from q to q'.

Let us illustrate this type of representation using the token ring example. Each process can be in one of two states, N or T, where N denotes that the process does not have the token, and T denotes that the process has the token. As the set of configurations Γ we take the set $\{N, T\}^*$ and use TN^* as the set of initial configurations, in which the leftmost process has the token, and as the transition relation we take

$$Id_{N^*} \cdot [(T, N) \cdot (N, T)] \cdot Id_{N^*} \ \cup \ Id_{N^*} \cdot Id_{\{N,T\}} \cdot Id_{N^*},$$

the union of two relations, of which the first denotes the passing of the token from a process to its right neighbor, and the second denotes an idling computation step.

Note that the transition relation is implicitly constrained by the invariant that there is exactly one token in the system. To model a ring, we have to handle the case where the process with the highest index passes the token to the process with the lowest process, connecting the ring. The transition relation handling this case can be given as

$$[(N,T)] \cdot Id_{N^*} \cdot [(T,N)]$$

4.2 INTEGER VARIABLES

Consider a system with integer variables $x_1, x_2, \ldots, x_n$. To model this as a regular model, we associate with each integer variable x_i a boolean variable b_i. As the alphabet we take the set $\Sigma = \{true, false\}^n$. Each state is represented by a word over Σ such that the boolean variable b_i is true at position j iff $x_i = j$.

This way of representing integer variables is used by MONA [HJJ+96] when translating monadic second order logic into automata. The variables of this logic actually range over the finite subsets of natural numbers. In terms of our notation, using a boolean variable b, we would represent a subset of natural numbers with a word of length n where i is in the set iff $i < n$ and b is true at position i. Note that since we are representing a *finite* subset we can always find an word that is long enough to represent the subset we are interested in.

Using this representation, we can represent various constraints on the integer variables. For example, for two variables x and y, we can represent $x < y$, $x \leq y$, and $x = y + c$ for any constant c.

The operation $x := y + c$ can be represented by the regular relation

$$\left[(y = y')^*\right] \cap \left[(\neg x' \ \wedge \ \neg y)^* \cdot (\neg x' \ \wedge \ y) \cdot (\neg x' \ \wedge \ \neg y)^{c-1} \cdot (x' \ \wedge \ \neg y) \cdot (\neg x' \ \wedge \ \neg y)^*\right]$$

4.3 QUEUES

Consider a system with a queue containing a set of messages M, where M^* represents the queue content as the set of configurations. Since regular relations are length preserving, we have to add a padding symbol $\perp$ to handle operations that change the length of the queue. The symbol $\perp$ represents an empty slot that can be filled in when for example a message is sent to the queue. Similarly, a message that is received from the queue is converted back to $\perp$.

Suppose that we want to represent a system consisting of a finite control and a FIFO queue. Let Q denote a finite set of control states and M denote a finite set of messages in the queue. As the alphabet, we take the union of the set of control states and the set of messages, and add a symbol $\perp$ to represent an empty slot, i.e., the alphabet is $\Sigma = Q \cup (M \cup \{\perp\})$. Each state in the system is represented by a word in $Q\perp^* M^* \perp^*$ where the first position represent the control state and

where the rest of the word represents the content of the queue. The symbol $\perp$ represents empty slots and is used to allow the queue to grow and shrink. Note that each configuration is of a certain length and thus represents a system with a fixed number of positions available for the queue. When performing verification, however, we will verify the system for all the possible lengths. Thus, even if some transition can "get stuck" because there is no empty slot to fill in, there will always be an instance with a larger number of positions where there is an empty slot available.

To send a message $m \in M$ when in a control state $q \in Q$, we use the regular relation $[(q,q)] \cdot Id_{\perp^*} \cdot Id_{M^*} \cdot [(\perp, m)] \cdot Id_{\perp^*}$. To receive a message $m \in M$ when in a control state $q \in Q$, we use the regular relation $[(q,q)] \cdot Id_{\perp^*} \cdot [(m, \perp)] \cdot Id_{M^*} \cdot Id_{\perp^*}$.

4.4 STACKS

Stacks can be modeled in a similar way as queues. Let Q denote a finite set of control states and let M denote the set of stack contents. As the alphabet, we take $\Sigma = Q \cup (M \cup \{\perp\})$ where $\perp$ is a symbol used to represent empty slots. A state of the system is then represented by a word in $Q \cdot M^* \cdot \perp^*$ where the first position is used to represent the control state, and the rest of the word is used to represent the contents of the stack.

Pushing m on the stack while in control state q can be represented by the regular relation $[(q,q)] \cdot Id_{M^*} \cdot [(\perp, m)] \cdot Id_{\perp^*}$ and popping m off the stack while in control state q can be represented by the regular relation $[(q,q)] \cdot Id_{M^*} \cdot [(m, \perp)] \cdot Id_{\perp^*}$.

Chapter 5

Model Checking

To model check a system, we take a model of the system and a specification and check that all behaviors of the model are behaviors of the specification. In this chapter, we will explain how to automate this process in the case of a regular model. We will reduce the problem into a problem of checking emptiness of an expression over regular relations, a problem discussed in Chapter 6.

5.1 THE MODEL CHECKING PROBLEM

We begin by defining the *model checking problem*.

Instance: A set of configurations Γ, a model $\mathcal{M}$ over Γ, and a temporal formula $\varphi \in PTL(\Gamma)$ over Γ.

Question: $\mathcal{M} \models \varphi$?

We begin with the case that φ is a safety property, in which case we solve this problem using *reachability analysis*, checking whether a model can reach a particular configuration. Then we present a method for the general case.

Let Γ be a set of configurations and let $\varphi \in PTL(\Gamma)$ be a safety property. Since φ is a safety property, there is a model $\mathcal{M}_\varphi$ over $\Gamma \times \Gamma'$ for a set of configurations Γ' such that $\pi([\![\mathcal{M}_\varphi]\!]) = [\![\varphi]\!]$ where π is the projection defined by $\pi(\gamma, \gamma') = \gamma$, and a set of configurations $\Gamma_0 \subseteq \Gamma \times \Gamma'$ such that $[\![\varphi]\!] = \pi({\Gamma_0}^\omega)$. Thus, to check for runs of φ we can check for runs of $\mathcal{M}_\varphi$ only visiting configurations in Γ_0. We can think of Γ_0 being configurations which are good and the rest begin configurations which are bad. Let $\mathcal{M} = (\Gamma^I, \longrightarrow, \Gamma^F)$ be a model over Γ and suppose that we want to check whether $\mathcal{M} \models \varphi$. We assume that $\Gamma^F = \Gamma$, which is true in a simple model without fairness conditions. Let $\mathcal{M}'$ be the model over $\Gamma \times \Gamma'$ obtained from $\mathcal{M}$ in such a way that it functions like $\mathcal{M}$ except that it ignores the second component from Γ'. We know that $[\![\varphi]\!]$ equals $\pi({\Gamma_0}^\omega)$, and therefore, if we can calculate the set of configurations reachable by $\mathcal{M}'$, i.e. configurations that can occur in any run of $\mathcal{M}'$, we can establish whether $\mathcal{M} \models \varphi$ by checking if this set is a subset of Γ_0. The set of *reachable configurations* of $\mathcal{M}$ can be written as the image $\longrightarrow^*(\Gamma^I)$, and can be calculated in many ways.

Reachability Analysis A naive reachability analysis works as follows. A set of configurations is maintained, initialized to the set of initial configurations. Iteratively, we add new configurations to this set by applying the transition relation to the set of configurations we have reached so far. This process goes on until no new configuration is found.

Let us illustrate reachability analysis with the token ring example. The initial set of configurations in this system is given by the regular expression $T \cdot N^*$, which is the set we start with. Let $\longrightarrow$ denote the relation between configurations such that one process passes the token to the right. Applying this relation to our set of configurations gives us the set $\longrightarrow (T \cdot N^*) = N \cdot T \cdot N^*$, which is added to the current set of configurations. After m iterations, we get the set $\{N^{l_1} \cdot T \cdot N^{l_2} \ : \ 0 \leq l_1 \leq m \ \wedge \ l_2 \geq 0\}$. Clearly, in this case the analysis will not terminate, since this set is different for every m.

There are several strategies to make the reachability analysis terminate, by enhancing the analysis with transition relations that correspond to applying transitions several times, thus reaching the set of reachable states more quickly. These types of transition relations are called *meta transitions* by Boigelot [Boi98] and is defined as any transition relation which is a subset of the transitive closure of the transition relation defining the program. Thus, such a transition represent a subset of the reachability relation between the states. Thus, when applying such a meta transition in the reachability analysis we can only get reachable states, with the difference that it may take fewer iterations of the analysis to reach a particular state. In the example with token ring, a meta transition that could be added is the transitive closure $\longrightarrow^*$, which corresponds to passing the token to the right an arbitrary number of times.

Computing the image $\longrightarrow^* (\Gamma^I)$ directly is sometimes not feasible. The relation $\longrightarrow$, however, is often a union of several relations, each representing some operation in the system. Suppose that there are relations $\longrightarrow_1, \longrightarrow_2, \ldots, \longrightarrow_n$ such that $\longrightarrow = \bigcup_{1 \leq i \leq n} \longrightarrow_i$. We can then choose to perform the reachability analysis by, instead of applying $\longrightarrow^*$, applying $\longrightarrow_i^*$ for some i with $1 \leq i \leq n$. At each iteration, we choose a different i. Hopefully, the relations $\longrightarrow_i^*$ are easier deal with than $\longrightarrow^*$.

Another approach is to encode the reachability analysis into a composition of regular relations and check for emptiness. Using Γ^I to denote the set of initial configurations, $\longrightarrow$ to denote the transition relation, and Γ_b to denote a set of bad configurations that we want to check if we can reach, the following composition is nonempty iff Γ_b can be reached:

$$Id_{\Gamma^I} \circ \longrightarrow^* \circ Id_{\Gamma_b}$$

Common to all these approaches is that they involve a computation of *non-finite compositions* of relations. This is a topic that we will discuss in Chapter 6, where a semi-algorithm is given, computing an automaton representing a (possibly non-finite) composition of regular relations.

There is another approch for improving reachability analysis that works by after a few iterations looking at the sequence of sets of configurations obtained in each iteration and trying to guess the limit. This can be referred to as *widening-based* techniques, named after a technique called widening used in abstract interpretation [CC77], and was presented in [BJNT00] but is not a part of this thesis.

5.2 MODEL CHECKING GENERAL PROPERTIES

With the case for safety properties in mind, we now turn to the general case. Suppose that we want to check whether $\mathcal{M} \models \varphi$ for a model $\mathcal{M}$ and a property φ. We can equivalently check the emptiness of the set $[\![\mathcal{M}]\!] \cap [\![\neg\varphi]\!]$. We saw in Chapter 2 that for every temporal formula there is a model with the same set of runs except for an extra component added to the set of configurations used for the translation. Thus the problem becomes a problem of checking intersections of runs of models.

Instance: A set of configurations Γ and a set of models $\{\mathcal{M}_i\}_{i\in I}$.

Question: $\bigcap_{i\in I} [\![\mathcal{M}_i]\!] = \emptyset$?

To illustrate this reduction, suppose that we have a model $\mathcal{M}$ and a property φ over a set of configurations Γ and that we like to verify that $\mathcal{M} \models \varphi$. Using Theorem 2.5, we obtain a model $\mathcal{M}_\varphi$ over $\Gamma \times \Gamma'$ for a set of configurations Γ' such that $\pi([\![\mathcal{M}_\varphi]\!]) = [\![\varphi]\!]$ where π is the projection defined by $\pi(\gamma, \gamma') = \gamma$. Then we can check whether the set $[\![\mathcal{M}]\!] \cap \pi([\![\mathcal{M}_\varphi]\!])$ is empty. As noted in the previous section, the model $\mathcal{M}$ can be extended to a model over $\Gamma \times \Gamma'$ by ignoring the second component. For an extended $\mathcal{M}$, we have to check whether the set $[\![\mathcal{M}]\!] \cap [\![\mathcal{M}_\varphi]\!]$ is empty.

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two models over a set of configurations Γ. To check whether the set $[\![\mathcal{M}_1]\!] \cap [\![\mathcal{M}_2]\!]$ of runs is empty we need to find runs belonging to both $\mathcal{M}_1$ and $\mathcal{M}_2$. A new model $\mathcal{M}$ can created with the property that $[\![\mathcal{M}]\!] = [\![\mathcal{M}_1]\!] \cap [\![\mathcal{M}_2]\!]$, an example of this is given later in this section. Then we use the observation that the transition relation of a model is length preserving, which means that the set of configurations occurring in a run of a program will be finite. This implies that checking for runs of a model reduces to checking for loops.

Theorem 5.1 *Let $\mathcal{M} = (\Gamma^I, \longrightarrow, \Gamma^F)$ be a regular model over a set of configurations Γ over a finite alphabet Σ. Then there is a run $\theta \in [\![\mathcal{M}]\!]$ iff there is a*

sequence $\gamma_0\gamma_1 \cdots \gamma_m \cdots \gamma_{m'}$ *such that* $\gamma_m = \gamma_{m'} \in \Gamma^F$ *and* $(\gamma_j, \gamma_{j+1}) \in \longrightarrow$ *for all* $j < m'$.

Proof: Let $\theta \in [\![\mathcal{M}]\!]$ be an accepting run (where configurations are of equal length, say n). Since the set of configurations over Σ of length n is finite, there must be some m, m' with $m' > m$ such that $\gamma_m = \gamma_{m'}$.

Conversely, let $\gamma_0\gamma_1 \cdots \gamma_m \cdots \gamma_{m'}$ be a sequence of configurations from Γ such that $\gamma_m = \gamma_{m'} \in \Gamma^F$ and $(\gamma_i, \gamma_{i+1}) \in \longrightarrow$ for all $i < m'$. It follows that $\gamma_0\gamma_1 \cdots \gamma_m\gamma_{m+1}\gamma_{m+2} \cdots \gamma_{m'}\gamma_{m+1}\gamma_{m+2} \cdots \gamma_{m'} \cdots$ is in $[\![\mathcal{M}]\!]$. □

Using the above theorem, we can check whether the set $[\![\mathcal{M}]\!]$ of runs is empty by looking for loops in $\mathcal{M}$ containing a reachable configuration that is in the set of accepting configurations. In the case of a safety property, the set of accepting configurations is the set of configurations where some bad configuration has been seen. Thus, for safety properties, we are looking for loops in the model where a reachable bad configuration has occurred.

The problem of finding loops can be reduced checking the emptiness of a composition of regular relations as follows. For a model $(\Gamma^I, \longrightarrow, \Gamma^F)$, the set of loops beginning and ending in a reachable state is given by the relation $Id_{\longrightarrow^*(\Gamma^I)} \cap \longrightarrow^+$, containing pairs containing the same configuration and where there is a path of at least length one from the configuration back to itself.

Using this method on a model $\mathcal{M}_1 = (\Gamma_1^I, \longrightarrow_1, \Gamma)$ representing the system where the set of accepting configurations is the set of all configurations, and a model $\mathcal{M}_2 = (\Gamma_1^I, \longrightarrow_2, \Gamma_2^F)$ representing a property, we can check for loops that are in both models by checking emptiness of the relation

$$Id_{(\longrightarrow_1 \cap \longrightarrow_2)^*(\Gamma_1^I \cap \Gamma_2^I) \cap \Gamma_2^F} \cap (\longrightarrow_1 \cap \longrightarrow_2)^+$$

5.3 FAIRNESS

When verifying protocols, one wants to add *fairness* conditions constraining the possible runs in a system. Fairness conditions are assumptions on the dynamic behavior of a system. One example of a fairness condition is a condition stating that no process can idle indefinitely. This is normally true because there is a scheduler in the system that does not let any process starve.

Fairness conditions are handled by stating the condition as a property φ_f such that the property holds for desirable runs. Then instead of checking $\mathcal{M} \models \varphi$ for a property φ and a model $\mathcal{M}$, the condition $\mathcal{M} \models \varphi_f \implies \varphi$ is checked. This says that for all runs of $\mathcal{M}$, *if* φ_f holds for this run, *then* φ is true. If φ_f is not true, we do not care since we assume that the system can not have such a run.

5.4 PARAMETERIZED PROPERTIES

Properties are specified using models with regular relations and regular sets, which makes it possible to handle a parameterized number of properties, for some conditions.

Let φ_i be a property parameterized by i. These properties can be encoded into another property as follows. We associate with each position i in the configurations an observer variable b which is set to true when the program reaches a configuration in the set of accepting configurations for φ_i. When b is true is all positions it is set to false again in all positions. The set of accepting configurations is then a configuration where b is true at all positions.

Chapter 6

Regular Compositions

In this chapter, we investigate compositions of regular relations including the reflexive and transitive closure operator *. To reason about programs, we want to reason about its dynamic behavior, and this is related to the relational composition of relations. We give a method to compute an automaton recognizing a composition called a *regular composition* of a given set of relations. Regular compositions can encode the verification problems we are interested in, including reachability analysis and the general model checking problem. This includes the reflexive and transitive closure R^* of a regular relation R. It was noted in Chapter 3 that there are regular relations R such that R^* is not regular. Thus, our method is in general incomplete.

6.1 COLUMN TRANSDUCER

A regular relation can be represented by an automaton over pairs of an alphabet. Such an automaton is commonly referred to as a *transducer*, which is the term we will use in this chapter. Our idea is based on the observation that each set of states of a transducer represents a regular relation, given by the set of suffixes of this state. Thus, we can identify regular relations by states in a transducer. We will define operations on states that correspond to the operations $\cup$, $\circ$ and * on relations. We begin with the union operation.

Theorem 6.1 *Let $A = (Q, S, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ for some alphabet Σ. Then for all $Q_0, Q_1 \subseteq Q$, $[\mathsf{suff}(Q_0)] \cup [\mathsf{suff}(Q_1)] = [\mathsf{suff}(Q_0 \cup Q_1)]$.*

Proof: Follows directly from the definition of $\mathsf{suff}()$ and $[]$. □

Thus, we can use union of subsets of transducer states to represent the union of regular relations. We will extend this idea to $\circ$ and *, by considering sequences of transducer states.

Let us first explore what it means that a pair of words (w, w') is in the relation $[\mathsf{suff}(q)]$ for a state q. Let $A = (Q, S, \Delta, F)$ be a transducer. For a

state $q \in Q$, a pair of words (w, w') is in $[\mathsf{suff}(q)]$ iff there is a run $q_0 q_1 \cdots q_n$ of A whose first state is q and whose final state is in F, accepting the word $w \times w' = (a_1, a'_1)(a_2, a'_2) \cdots (a_n, a'_n)$. If we let $q \xrightarrow{(a,a')} q'$ denote that $\Delta(q, (a, a'), q')$ holds, we can depict this run below where $q_0 = q$ and $q_n \in F$

$$q_0 \xrightarrow{(a_1,a'_1)} q_1 \xrightarrow{(a_2,a'_2)} \cdots \xrightarrow{(a_n,a'_n)} q_n$$

Now consider a sequence of states $q^1 q^2 \cdots q^m \in Q^*$ of length m. We want to use this sequence to represent the relation $\big[\mathsf{suff}(q^1)\big] \circ \big[\mathsf{suff}(q^2)\big] \circ \cdots \circ [\mathsf{suff}(q^m)]$. Let us explore what is means for a pair of words to be in this relation. A pair of words (w, w') is in $\big[\mathsf{suff}(q^1)\big] \circ \big[\mathsf{suff}(q^2)\big] \circ \cdots \circ [\mathsf{suff}(q^m)]$ iff there are words $w^0, w^1, \ldots, w^m$ such that $w = w^0$, $w' = w^m$, and $(w^{j-1}, w^j) \in \big[\mathsf{suff}(q^j)\big]$ for all i with $0 < j \leq m$. Let w^i be the word $a^j_1 a^j_2 \cdots a^j_n$ of length n. Then $(w^{j-1}, w^j) \in [\mathsf{suff}(q_j)]$ means that there is a run $q^j_0 q^j_1 \cdots q^j_n$ of A where the first state q^j_0 is q^j and the last state q^j_n is in F, accepting the word $(a^{j-1}_1, a^j_1)\ (a^{j-1}_2, a^j_2) \cdots (a^{j-1}_n, a^j_n)$. These runs can be organized into a matrix as follows:

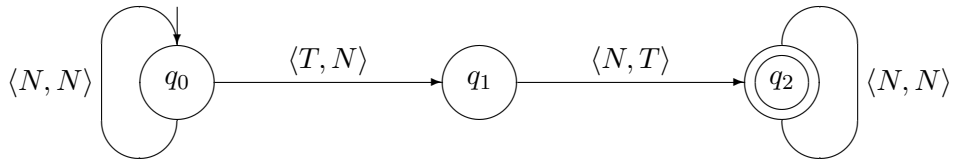
$$\begin{array}{ccccccccc}
q^1_0 & \xrightarrow{(a^0_1,a^1_1)} & q^1_1 & \xrightarrow{(a^0_2,a^1_2)} & q^1_2 & \cdots & q^1_{n-1} & \xrightarrow{(a^0_n,a^1_n)} & q^1_n \\
q^2_0 & \xrightarrow{(a^1_1,a^2_1)} & q^2_1 & \xrightarrow{(a^1_2,a^2_2)} & q^2_2 & \cdots & q^2_{n-1} & \xrightarrow{(a^1_n,a^2_n)} & q^2_n \\
\vdots & & \vdots & & & & \vdots & & \\
q^m_0 & \xrightarrow{(a^{m-1}_1,a^m_1)} & q^m_1 & \xrightarrow{(a^{m-1}_2,a^m_2)} & q^m_2 & \cdots & q^m_{n-1} & \xrightarrow{(a^{m-1}_n,a^m_n)} & q^m_n
\end{array}$$

with m rows, where each row shows a run of A with n transitions.

Let us illustrate this with the token ring example. Let $\Sigma = \{N, T\}$ be the alphabet and let $A = (\{q_0, q_1, q_2\}, \{q_0\}, \Delta_t, \{q_2\})$ be an automaton over $\Sigma \times \Sigma$ where

$$\Delta_t = \{(q_0, (N, N), q_0), (q_0, (T, N), q_1), (q_1, (N, T), q_2), (q_2, (N, N), q_2)\},$$

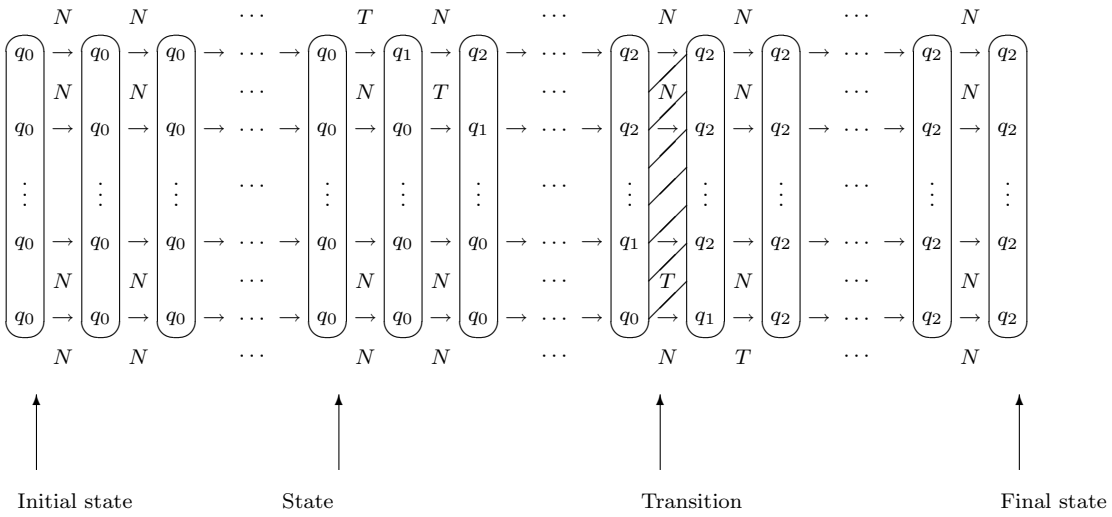
denoting the automaton depicted below, representing the transition relation where the token is passed to the right.



The matrix representing runs where the token is passed from the left to the right m times is given as follows.

$$
\begin{array}{ccccccccccccc}
q_0 & \xrightarrow{(T,N)} & q_1 & \xrightarrow{(N,T)} & q_2 & \xrightarrow{(N,N)} & q_2 & \cdots & q_2 & \xrightarrow{(N,N)} & q_2 & \xrightarrow{(N,N)} & q_2 \\
q_0 & \xrightarrow{(N,N)} & q_0 & \xrightarrow{(T,N)} & q_1 & \xrightarrow{(N,T)} & q_2 & \cdots & q_2 & \xrightarrow{(N,N)} & q_2 & \xrightarrow{(N,N)} & q_2 \\
 & \vdots & & \vdots & & \vdots & & & & \vdots & & \vdots & \\
q_0 & \xrightarrow{(N,N)} & q_0 & \xrightarrow{(N,N)} & q_0 & \xrightarrow{(N,N)} & q_0 & \cdots & q_1 & \xrightarrow{(N,T)} & q_2 & \xrightarrow{(N,N)} & q_2 \\
q_0 & \xrightarrow{(N,N)} & q_0 & \xrightarrow{(N,N)} & q_0 & \xrightarrow{(N,N)} & q_0 & \cdots & q_0 & \xrightarrow{(T,N)} & q_1 & \xrightarrow{(N,T)} & q_2
\end{array}
$$

Now, we observe that the columns of the above matrix can be seen as states in another automaton where the transition relation is given by a relation between columns. A run of this new automaton for the token ring example is given below, where we have pointed out the leftmost column to be the initial state, the rightmost column to be the final state, and the transition relation to be a relation between to columns.



In this new automaton, we see that to represent the relation $\left[\mathsf{suff}(q^1)\right] \circ \left[\mathsf{suff}(q^2)\right] \circ \cdots \circ \left[\mathsf{suff}(q^m)\right]$, we can use the sequence $q^1 q^2 \cdots q^m$ which denotes a column in the automaton such that $\mathsf{suff}(q^1 q^2 \cdots q^m) = \left[\mathsf{suff}(q^1)\right] \circ \left[\mathsf{suff}(q^2)\right] \circ \cdots \circ \left[\mathsf{suff}(q^m)\right]$. This means that we can now represent the operator $\circ$ by using the concatenation operator on sequences of states. By generalizing this new automaton to have states that are sequences of arbitrary length, we can also handle the operator *.

Let us define what we mean by this new automaton more precisely. For a transducer $A = (Q, S, \Delta, F)$ let the *column transducer* for A be defined as the automaton

$\widehat{A} = (Q^*, S^*, \widehat{\Delta}, F^*)$ where $\widehat{\Delta}(q_1 q_2 \cdots q_n, (a, a'), q'_1 q'_2 \cdots q'_n)$ holds for sequences of equal length n such that there exist $a_0, a_1, \ldots, a_n$ with $a = a_0$, and $a' = a_n$, and $\Delta(q_i, (a_i, a_{i+1}), q_{i+1})$ holds for all i with $0 \leq i < n$.

We can now, using a column transducer, express all the operations we are interested in.

Theorem 6.2 *Let $A = (Q, S, \Delta, F)$ be a transducer and let $\widehat{A} = (Q^*, S^*, \widehat{\Delta}, F^*)$ be the column automaton for A. Then, for two sets of sequences $X_1, X_2 \subseteq Q^*$, the following hold:*

1. $[\mathsf{suff}(X_1)] \cup [\mathsf{suff}(X_2)] = [\mathsf{suff}(X_1 \cup X_2)]$
2. $[\mathsf{suff}(X_1)] \circ [\mathsf{suff}(X_2)] = [\mathsf{suff}(X_1 \cdot X_2)]$
3. $[\mathsf{suff}(X_1)]^* = [\mathsf{suff}(X_1^*)]$

Proof: (1) follows directly from the definition of $\mathsf{suff}()$. To prove one direction of (2), let $(w, w') \in [\mathsf{suff}(X_1)] \circ [\mathsf{suff}(X_2)]$. Then there exist a word w'' such that $(w, w'') \in [\mathsf{suff}(X_1)]$ and $(w'', w') \in [\mathsf{suff}(X_2)]$. Then $w \times w'' \in \mathsf{suff}(X_1)$ and $w'' \times w' \in \mathsf{suff}(X_2)$. Thus, there exist $x_1 \in X_1$ and $x_2 \in X_2$ such that $w \times w'' \in \mathsf{suff}(x_1)$ and $w'' \times w' \in \mathsf{suff}(x_2)$. Further, we have that $\widehat{\Delta}(x_1, w \times w'') \cap F^* \neq \emptyset$ and $\widehat{\Delta}(x_2, w'' \times w') \cap F^* \neq \emptyset$ hold. It follows from the definition of $\widehat{\Delta}$ that $\widehat{\Delta}(x_1 x_2, w \times w') \cap F^* \neq \emptyset$ holds. Thus, $w \times w' \in \mathsf{suff}(x_1 x_2)$. Since $x_1 \in X_1$ and $x_2 \in X_2$, we have that $x_1 x_2 \in X_1 X_2$, and thus it follows that $w \times w' \in \mathsf{suff}(X_1 X_2)$ and $(w, w') \in [\mathsf{suff}(X_1 X_2)]$. The other direction can be proved in a similar way.

To prove (3), use induction using (2). □

6.2 SUBSET CONSTRUCTION

The column transducer defined in the previous section has an infinite number of states since the columns can be of arbitrary length. We will now attack the problem of making the column transducer finite-state. Our strategy will be to apply the standard subset construction (for a description of the subset construction, see e.g. Kozen[Koz97] ch. 6).

For an automaton $A = (Q, S, \Delta, F)$, the result of the subset construction is the automaton $(2^Q, \{S\}, \Delta_P, \{F_0 : F_0 \cap F \neq \emptyset\})$, where Δ_P is defined such that $\Delta_P(Q_0, a, Q'_0)$ iff $\Delta(Q_0, a) = Q'_0$. It is constructed by an iterative process, starting with the set of initial states S applying Δ obtaining new states $\Delta(S, a)$ for each symbol a. The process is repeated until no new states are found.

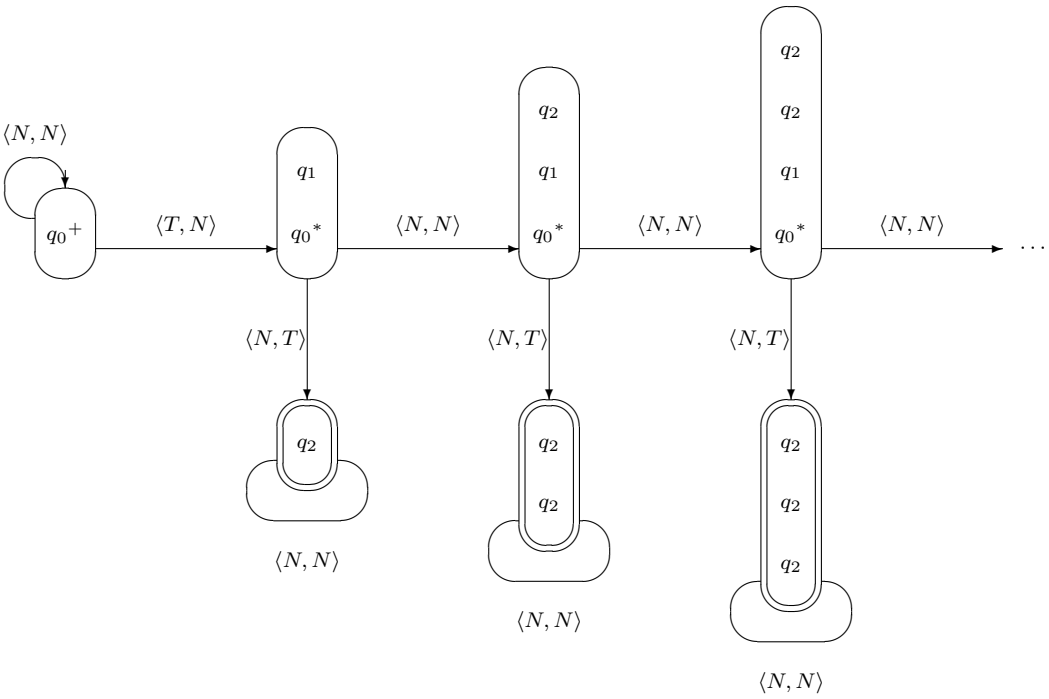
Let us apply the subset construction on a column transducer. For the remainder of this section, let $A = (Q, S, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ and let $\widehat{A} =$

$(Q^*, S^*, \widehat{\Delta}, F^*)$ be a column transducer for A. To apply the subset construction, we need to compute the image $\widehat{\Delta}(X, (a, a'))$ for a set of columns, i.e., states in the column transducer $X \subseteq Q^*$, and some symbols $a, a' \in \Sigma$. In general, the transition relation is regular for regular sets of columns, due to the following theorem.

Theorem 6.3 *Let $a, a' \in \Sigma$ be two symbols. Then $\widehat{\Delta}_{(a,a')} = \{(x, x') \ : \ \widehat{\Delta}(x, (a, a'), x')\}$ is a regular relation on $Q^* \times Q^*$.*

Proof: The relation $R_1 = [\{(q, q', b, b') \ : \ (q, (b, b'), q') \in \Delta\}^*]$ is a regular relation on $Q^* \times Q^* \times \Sigma^* \times \Sigma^*$. The relation R_2 on $\Sigma^* \times \Sigma^*$ consisting of all words $(a_1 a_2 \cdots a_n, a'_1 a'_2 \cdots a'_n)$ such that $a = a_1$, $a' = a_n$, and $a_i = a'_{i-1}$ for all i with $1 < i \leq n$ is also regular. Now $\widehat{\Delta}_{(a,a')} = \pi_{(1,2)}(R_1 \ \cap \ (Q^* \overline{\times} Q^* \overline{\times} R_2))$. □

With this operation, we can apply the standard subset construction, by representing sets of states, i.e., sets of columns, using regular sets over the alphabet Q. Continuing our token ring example, we take the initial set of states of the transitive closure of the transition relation, which is the set q_0^+, a regular set over $\{q_0, q_1, q_2\}$, denoting one or more applications of the transition relation represented by the state q_0. The resulting construction is shown below.



6.3 DETECTING EQUIVALENT STATES

Applying the subset construction still does not yield a finite-state transducer in most cases. The reason is that there may be an infinite number of different states which are equivalent in the sense that they have the same set of suffixes. In this section, we will try to remedy this by attempting to find equivalent states in the same way as when minimizing an automaton.

Looking at the subset construction resulting from the token ring example, we can see that sequences of the state q_2 is growing in length. The relation denoted by the state q_2 is just checking the condition that all symbols are N, and it does not matter how many times this is checked as long as it is done at least once. This leads to the following simple observation.

Lemma 6.4 *Let R be an idempotent relation, i.e., $R = R \circ R$. Then $R^i = R^+$ for all $i > 0$.*

Proof: By induction on the number of compositions. □

Another observation is that since $[\mathsf{suff}(q_2)]$ is a subset of the identity relation, for any columns $x, x' \in Q^*$ we have that $[\mathsf{suff}(x \cdot q_2 \cdot x')] \subseteq [\mathsf{suff}(x \cdot x')]$. In general, this holds for any relation which is a subset of the identity relation.

Theorem 6.5 *Let Γ be a set configuartions and let R be a relation such that $R \subseteq Id_\Gamma$. Then $R \circ R = R$.*

Proof: Immediate. □

Corollary 6.6 *For a set of states $X \subseteq Q^*$ and a state $q \in Q$ such that $[\mathsf{suff}(q)] \subseteq Id_{\Sigma^*}$, the following hold:*

1. *If $x_1 \cdot q \cdot q \cdot x_2 \in X$, then $\mathsf{suff}(X) = \mathsf{suff}(X \cup \{x_1 \cdot q \cdot x_2\})$.*
2. *If $x_1 \cdot x_2 \in X$, then $\mathsf{suff}(X) = \mathsf{suff}(X \cup \{x_1 \cdot q \cdot x_2\})$.*

□

Based on the observations above, we will devise a method for detecting equivalent states. The method is based on *saturation*, which means that given two sets of columns, we try to add columns to both sets without changing the relations represented by the sets, and finally check if the two saturated sets are equal.

For a set of states X, the saturated set of X by a the set of states Q_0, denoted $\lceil X \rceil_{Q_0}$, is the set of states containing X and closed under the following rules for all states $q \in Q_0$:

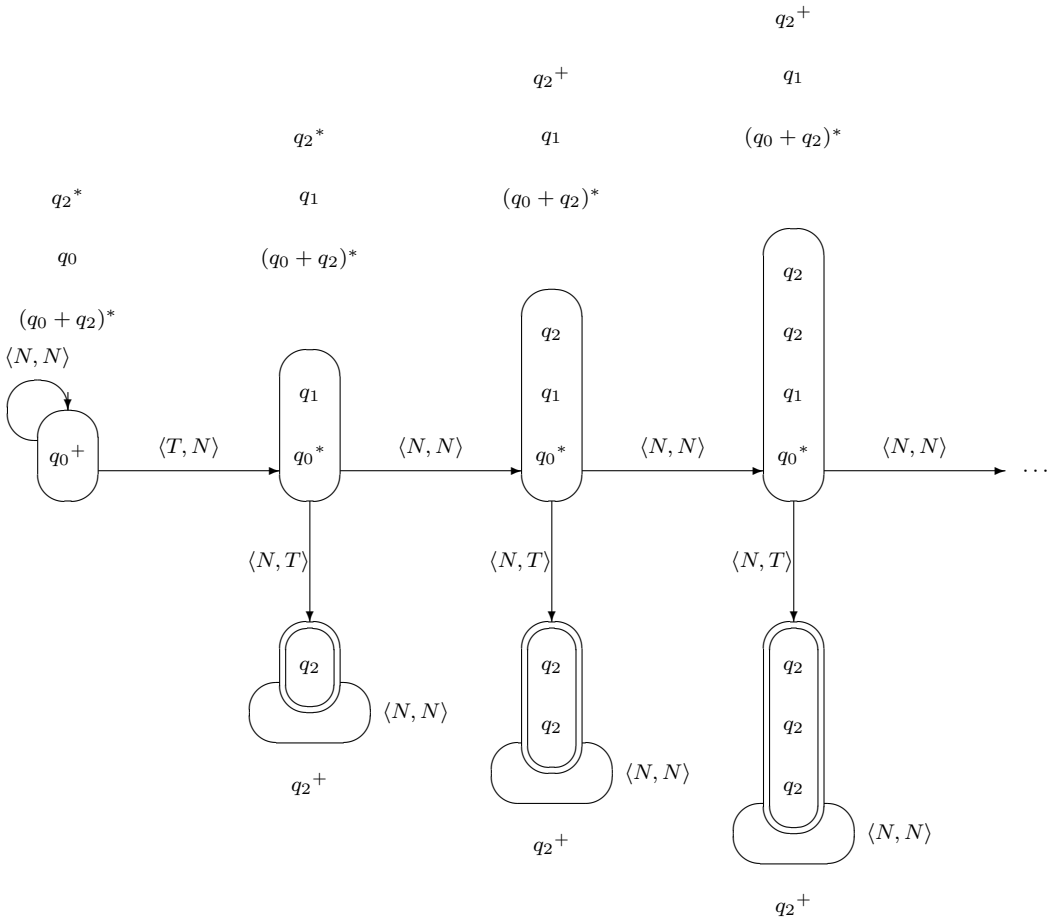
1. If $x_1 \cdot q \cdot q \cdot x_2 \in \lceil X \rceil_{Q_0}$, then $x_1 \cdot q \cdot x_2 \in \lceil X \rceil_{Q_0}$.
2. If $x_1 \cdot x_2 \in \lceil X \rceil_{Q_0}$, then $x_1 \cdot q \cdot x_2 \in \lceil X \rceil_{Q_0}$.

For a set X of sequences, we will denote by $\lceil X \rceil$ the set $\lceil X \rceil_{Q_0}$ where Q_0 is the set of states $q \in Q$ such that $[\mathsf{suff}(q)] \subseteq Id_{\Sigma^*}$. This is the saturation we will use and is motivated by the following theorem.

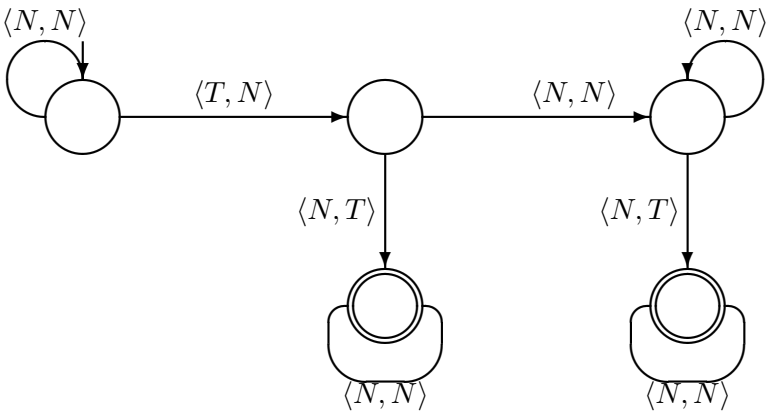
Theorem 6.7 *Let $X, Y \subseteq Q^*$ be sets of sequences. If $\lceil X \rceil = \lceil Y \rceil$, then $\mathsf{suff}(X) = \mathsf{suff}(Y)$.*

Proof: Follows from the rules of saturation and Corollary 6.6, in that the tuples in the relation represented by the added columns already exists in the relation represented by the columns that are already there. □

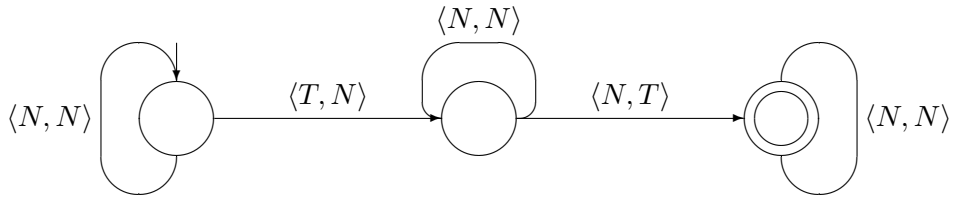
In the subset construction we use saturation in the following way. When we have computed a new state X, we check for each old state Y if it is equivalent by checking if $\lceil X \rceil = \lceil Y \rceil$. The result of saturating the states in the subset construction in the case of the token ring example is shown below:



We see that we do not get different saturated sets after some iterations, and thus the construction terminates resulting in the automaton shown below:



which is minimized using standard techniques to



6.4 SUFFICIENT CONDITIONS FOR TERMINATION

In this section we will give a sufficient condition for the termination of the algorithm of the previous section to compute a finite-state transducer for a regular composition. We do this by limiting the number of states in the columns from a special subset of the states, as follows.

Definition 6.8 Let $A = (Q, S, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ and let $X \subseteq Q^*$ be a set of sequences of automata states, and $Q_0 \subseteq Q$ be a subset of the states in A. The set of sequences X has *local depth* k w.r.t. Q_0 if all sequences $x \in X$ contains only at most k occurences of states from Q_0. □

The states Q_0 in the above theorem are the states that can only occur a bounded number of times in the columns. The rest of the states $Q \setminus Q_0$ will be handled by showing that any set of sequences of such states will be in one of a finite set of equivalence classes. These states will be restricted to be two states checking a condition of the form $Id_{\Sigma_0^*}$ for some $\Sigma_0 \subseteq \Sigma$. In terms of the token ring example, these states will be q_0 and q_2, and the state q_1 is the state that may only occur a

bounded number of times in the columns. For this transition, representing passing the token to the next process, the number of occurences of the state q_1 can be at most 1 since one position can only be in the middle of a change once, when having the token and passing it to the next process. The state q_2 will be saturated during the subset construction, and the state q_0 will already be saturated due to the following theorem.

Theorem 6.9 *Let $A = (Q, S, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ where $S = \{q_l\}$ for a state $q_l \in Q$ such that $[\mathsf{pref}(q_l)] \subseteq Id_{\Sigma^*}$ and let $\widehat{A} = (Q^*, S^*, \widehat{\Delta}, F^*)$ be the column automaton for A. Then for all $w \in (\Sigma \times \Sigma)^*$, we have that $\lceil \widehat{\Delta}(S, w) \rceil_{\{q_l\}} = \widehat{\Delta}(S, w)$*

Proof: Consider a sequence $x \in \widehat{\Delta}(S, w)$ for some word $w \in (\Sigma \times \Sigma)^*$. For any x_1, x_2 such that $x = x_1 \cdot x_2$ we have that $\mathsf{pref}(x_1 \cdot q_l \cdot x_2) \subseteq \mathsf{pref}(x_1 \cdot x_2)$, since $[\mathsf{pref}(q_l)] \subseteq Id_{\Sigma^*}$. From the fact that the set of initial states in the column transducer is q_l^* and that $x_1 \cdot x_2 \in \widehat{\Delta}(S, w)$, it follows that $x_1 \cdot q_l \cdot x_2 \in \widehat{\Delta}(S, w)$. Using a similar argument, we can show that if $x_1 \cdot q_l \cdot q_l \cdot x_2 \in \widehat{\Delta}(S, w)$, then $x_1 \cdot q_l \cdot x_2 \in \widehat{\Delta}(S, w)$. □

A state q_l such that $[\mathsf{pref}(q_l)] \subseteq Id_{\Sigma^*}$ is already saturated due to the above theorem, and a state q_r such that $[\mathsf{suff}(q_l)] \subseteq Id_{\Sigma^*}$ is saturated by our construction. The relations represented by combinations of such states can be divided into a finite set of equivalence classes, due to the following theorem.

Theorem 6.10 *Let $A = (Q, \{q_l\}, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ where q_l, q_r are states such that*

- $[\mathsf{pref}(q_l)] \subseteq Id_{\Sigma^*}$, *and*
- $[\mathsf{suff}(q_r)] \subseteq Id_{\Sigma^*}$

and let $\widehat{A} = (Q^*, S^*, \widehat{\Delta}, F^*)$ be the column automaton for A. Then the set of saturated sets $\lceil X \rceil_{\{q_l, q_r\}}$ such that $X \subseteq \{q_l, q_r\}^*$ is finite.

Proof: We prove that each of the saturated sets is a union of zero or more of

1. $\lceil \varepsilon \rceil_{\{q_l, q_r\}}$
2. $\lceil q_l \rceil_{\{q_l, q_r\}}$
3. $\lceil q_r \cdot q_l \rceil_{\{q_l, q_r\}}$
4. $\lceil q_l \cdot q_r \rceil_{\{q_l, q_r\}}$

5. $\lceil q_l \cdot q_r \cdot q_l \rceil_{\{q_l, q_r\}}$
6. $\lceil q_r \cdot q_l \cdot q_r \rceil_{\{q_l, q_r\}}$

To see this, note that each of the above sets represents constraints which is a combination of whether the sequences start and end with q_l or q_r, and whether it contains at least one q_l and/or q_r. Given these constraints, the saturation by $\{q_l, q_r\}$ inserts sequences such that all combinations is in the set. □

We immediately get the following result.

Corollary 6.11 *Let $A = (Q, \{q_l\}, \Delta, F)$ be a transducer over $\Sigma \times \Sigma$ where q_l, q_r are states such that*

- $[\mathsf{pref}(q_l)] \subseteq Id_{\Sigma^*}$, *and*
- $[\mathsf{suff}(q_r)] \subseteq Id_{\Sigma^*}$

and the column automaton $\widehat{A} = (Q^*, S^*, \widehat{\Delta}, F^*)$ for A has the property that

- *For all words $w \in (\Sigma \times \Sigma)^*$, the set of columns $\widehat{\Delta}(S, w)$ has local depth w.r.t. $Q \setminus \{q_l, q_r\}$.*

Then the set $\{\lceil \widehat{\Delta}(S, w) \rceil \ : \ w \in (\Sigma \times \Sigma)^*\}$ of saturated columns obtained during the subset construction of $\widehat{A}$ is finite.

Proof: Follows from the definition of local depth and Theorem 6.10. □

To better understand the above theorem, we will formulate it in terms of relations. First, we have to define what it means for a relation to have local depth.

Let Σ be an alphabet. A *guarded transition* is a triple (ϕ_L, R, ϕ_R) where ϕ_L and ϕ_R are regular languages over Σ and R is a regular relation on $\Sigma^* \times \Sigma^*$. We will associate with (ϕ_L, R, ϕ_R) the relation $Id_{\phi_L} \cdot R \cdot Id_{\phi_R}$. The separation into three components serves to structure a transition into a guarded rewriting rule, where R is the rewriting rule, and ϕ_L and ϕ_R are contexts in which the rewriting may occur. A set of indicies C is a *change set* w.r.t. a pair $(w, w') \in (\phi_L, R, \phi_R)$ and the guarded transition (ϕ_L, R, ϕ_R) if $w = w_l \cdot u \cdot w_r$ and $w' = w_l \cdot u' \cdot w_r$ where $w_l \in \phi_L$, and $w_r \in \phi_R$, and $(u, u') \in R$ and $C = \{i \ : \ |w_l| < i \leq |w_l \cdot u|\}$. A sequence $w_0 w_1 \cdots w_m$ of words of equal length n such that $(w_j, w_{j+1}) \in (\phi_L, R, \phi_R)$ for all j with $1 \leq j < m$ has *local depth k* w.r.t. (ϕ_L, R, ϕ_R) if there is a change set C_j w.r.t. to (w_j, w_{j+1}) and (ϕ_L, R, ϕ_R) for all j with $1 \leq j < m$ such that for all i with $1 \leq i \leq n$ we have that the set $\{j \ : \ i \in C_j\}$ has at most k elements. Finally, (ϕ_L, R, ϕ_R) has *local depth k* if for all pairs $(w, w') \in (\phi_L, R, \phi_R)^*$ there is a sequence $w_0 w_1 \cdots w_m$ with local depth k w.r.t. (ϕ_L, R, ϕ_R) such that $w = w_0$ and $w' = w_m$.

Theorem 6.12 *Let (ϕ_L, R, ϕ_R) be a guarded transition with local depth k for some k where $\phi_L = \Sigma_L^*$ and $\phi_R = \Sigma_R^*$ for some $\Sigma_L, \Sigma_R \subseteq \Sigma$. Then $(\phi_L, R, \phi_R)^*$ is regular.*

Proof: The relation (ϕ_L, R, ϕ_R) can be represented by a transducer having a state q_l such that $\mathsf{pref}(q_l) = \Sigma_L^*$, a state q_r such that $\mathsf{suff}(q_r) = \Sigma_R^*$, and states between q_l and q_r recognizing R. It follows from Corollary 6.11 that $(\phi_L, R, \phi_R)^*$ is regular. □

We can generalize the forms of the languages ϕ_L and ϕ_R in the above theorem to languages defined as follows.

Definition 6.13 Let Σ be an alphabet. A language L is a *left context* if there is some set of symbols $\Sigma_0 \subseteq \Sigma$ such that for all words $w \in L$ we have that $w \cdot w' \in L$ iff $w' \in \Sigma_0^*$. The set Σ_0 is called the *tail* of L. A language is a *right context* if its reverse language is a left context. □

Lemma 6.14 *Let Σ be an alphabet, L be a left context over Σ where Σ_L is the tail of L and let R be a regular relation on $\Sigma^* \times \Sigma^*$. Then $(Id_L \cdot R)^* = Id_L \cdot (Id_{\Sigma_L^*} \cdot R)^*$.*

Proof: Let $(w, w') \in (Id_L \cdot R)^*$. Let w_l be the shortest word in L such that $w = w_l \cdot w_0$ for some word w_0. Since w_l is shortest it follows that $w' = w_l \cdot w_0'$ for some word w_0'. From the definition of left context it follows that $(w_0, w_0') \in (Id_{\Sigma_L^*} \cdot R)^*$.

The other direction follows from the fact that $L \cdot \Sigma_L^* \subseteq L$. □

Theorem 6.15 [JN00] *Let (ϕ_L, R, ϕ_R) be a guarded transition with local depth k for some k where ϕ_L is a left context and ϕ_R is a right context Then $(\phi_L, R, \phi_R)^*$ is regular.*

Proof: Apply Lemma 6.14 twice followed by Theorem 6.12. □

Chapter 7

Implementation

In this chapter we describe an implementation of regular model checking to provide evidence that it is a framework for *automated* verification. It is important to stress that we do not claim to have the most efficient framework available for all types of systems. Even so, we are able to provide some promising results regarding the efficiency of our method.

Before the implementation, little was known about the applicability of the methods in practice. The most important tool in regular model checking is automata theory for representing regular sets, which made it probable that the efficiency of the methods were to rely heavy on the implementation of automata. There are several packages available for automata. For example, verification based on monadic second order logic use automata as its basis, implemented by the packages MONA[HJJ+96] and Mosel[KMMG97]. There is also a package called AMoRE[MMP+95], implementing most standard operations on automata. Our first implementation was based on MONA, which uses BDDs to implement deterministic automata very efficiently. In particular, the minimization operation is very efficient. We wanted, however, to experiment with the use of nondeterministic automata but still use BDDs as a basis for the implementation. Furthermore, the interface to the automata package in MONA is inspired by the use of monadic second order logic, while we needed a more direct interface. This lead to an implementation of a general package for nondeterministic automata based on BDDs, which has been used as a basis for the implementation of regular model checking.

A BDD is a data structure for representing relations over finite domains, exploiting regularities in the relations to keep the representation compact. MONA uses BDDs only for representing the alphabet part of the transition relation of the automata. As we use nondeterministic automata, we decided to represent the entire transition relation using BDDs, including the states. This has some interesting consequences, which we will discuss in this chapter.

We will begin by describing a framework for relations represented by BDDs. Although BDDs allow for a very compact representation of certain sets and for efficient operations it also introduces a degree of complexity. We will define a number

of abstractions to deal with this complexity, leading to a framework for working with relations represented by BDDs. This framework has been used as a basis for implementing the automata package, which allows for some interesting implementation techniques for some of the automata operations.

Once the automata package is in place, the implementation of regular model checking is quite straightforward. There are some tricks one can use, but most of the efficiency depends on the automata package.

7.1 RELATIONS - ABSTRACTING BDDS

There are many algorithms which are formulated using sets and relations, they are the basic elements of reasoning. Accordingly, there are also many data structures for representing sets and relations, for example binary trees, lists, and so on. One of these data structures are BDDs, *Binary Decision Diagrams*, used for representing large sets compactly by exploiting regularities in the set. Thus, for sets with a large degree of regularity this is a good choice.

Many algorithms are formulated with expressions like “For all x, ...” or “take the set of all ...”. With an explicit representation of the sets, this translates to loops over these sets. Using BDDs, we can use existential quantification instead which in some cases does not need to explicitly enumerate all the possible elements in the set due to the compact representation. The complexity of the algorithms then becomes dependent on the complexity of the sets, not their size.

In this section, we explore the possibilities the BDD representation can give us when representing relations. Having a generic way of representing relations also gives a way to cleanly implement the theories of automata and other frameworks, since they are formulated in terms of relations.

7.1.1 Binary Decision Diagrams

A binary decision diagram, or BDD for short, is a data structure used to represent boolean formulae or, equivalently, sets of bitstrings. This structure has been used extensively for verification of hardware circuits, due to its capability of representing large sets with a high degree of regularity. During the years, this has become a popular representation for different applications, including model checking of finite-state systems. An important property of BDDs is that they are *canonical*, i.e., they can be checked for equivalence in constant time. This property can be used to perform some operations on relations very efficiently.

A BDD is a tree where the nodes are labeled with boolean variables and the leaves are *true* and *false*. Each node has two edges, representing the two possible values of the variable labeling the node. A BDD represents a set of boolean variable assignments in the following way. Starting from the root, the edges are traversed

by looking at the variable labeling each node and the variable assignment. If the variable is true, we take the edge representing the true choice. If the variable is false, we take the edge representing the false choice. Eventually, we get to a leaf which is either true or false. If it is true, the variable assignment is in the set. If it is false, the variable assignment is not in the set.

Not all variables need to be represented in the BDD. If a variable is not present in the BDD, it means that the set is independent of the value of this variable. Also, isomorphic subtrees are shared which is the way in which BDDs exploits the regularities in the sets.

Formally, BDDs are defined as follows.

Definition 7.1 A *BDD* t is either a *leaf node* or a *variable node*. A leaf node is either *true* or *false*. A variable node is a tuple (v, t_l, t_h) where $v \in \mathcal{N}$ is a variable index and t_l, t_h are BDDs. We associate with each BDD t a relation on infinite bitstrings $0, 1^\omega$ such that $t(b)$ holds iff t is a the leaf node *true* or t is of the form (v, t_l, t_r) and we have that $t_l(b)$ holds if $b_v = 0$, and that $t_r(b)$ holds if $b_v = 1$. □

The nodes in a BDD are normally *ordered*, such that when traversing the tree from the root, we visit the variables in increasing order. We will assume that all BDDs are ordered. It is this property that makes BDDs *canonical*, i.e., there are no two BDDs representing the same relation.

BDDs are constructed using a *unique hashing table*, which ensures that identical BDDs are stored in the same position. This makes it possible to check for equivalence between two BDDs in constant time. The unique hashing table is used when constructing the BDD and is indexed by the variable and the two successors. BDDs are constructed bottom-up, consulting the unique table to see whether the BDD already exists.

7.1.2 Representing Relations

A BDD represents a set of bitstrings and will be used as the building block for representing relations. To represent relations, different parts of the bitstrings will be associated with different components of the relation. We will call such a part a *domain*.

A *domain* $D \subseteq \mathcal{N}$ is a set of variable indices. For a domain $D = \{i_0, i_1, \ldots\}$ where $i_0 < i_1 < \cdots$, and a bitstring $b \in \{0,1\}^\omega$, where we use $\{0,1\}^\omega$ to denote the set of infinite bitstrings, and a BDD t we associate with D a projection on bitstrings such that $D(b)_{i_k} = b_k$ for all k with $0 \leq k \leq |D|$. Note that the bitstring $D(b)$ is infinite iff D is infinite.

Using domains, we can *type* BDDs.

Definition 7.2 A *typed relation* R of arity k is a tuple $(\mathcal{D}, t)$ where $\mathcal{D} = (D_1, D_2, \ldots, D_k)$ is a tuple of *domains* where each $D_i \subseteq \mathcal{N}$ is a set of variable indices and t is a BDD. □

A typed relation $R = ((D_1, D_2, \ldots, D_k), t)$ defines a relation on tuples of bitstrings $\{0,1\}^{|D_1|} \times \{0,1\}^{|D_2|} \times \cdots \times \{0,1\}^{|D_k|}$ consisting of the set of tuples $(D_1(b), D_2(b), \ldots, D_k(b))$ such that $t(b)$ holds.

7.1.3 Exploiting the Structure of BDDs

When we have divided the BDDs into parts as we do when we type them using domains to specify variables that describe different components of the relations, we can exploit the structure of the BDDs to implement operations on relations. The common operations on relations such as intersection and union is part of the standard operations on BDDs, as is the projection operation corresponding to existential quantification. Here, we describe an operation for finding images of a binary relation which will be used to implement minimization of automata.

The first observation is that each subtree represents a set, since each subtree is also a BDD. When we have traversed the BDD from the root to a subtree, we have chosen values for the variables we have seen so far. If we think of the part that we have traversed as the first component of a binary relation, and the remaining part as the second component of a binary relation, and consider the result of applying the relation to the sets of bitstrings represented by the choices we have made so far, then the subtree we are currently at represents the image under this set of bitstrings.

This leads to a technique for *finding images* of a relation by finding the set of subtrees. This requires that the variable indices of the second component of the binary relation are higher than any variable index of the first component.

Let $R = ((D_1, D_2), t)$ be a typed binary relation such that all variable indices in D_2 are higher than any variable index in D_1. Then we can compute the set of images $\{R(x) \ : \ x \in \{0,1\}^{|D_1|}\}$ of R by finding subtrees in the BDD t that are located below any variables in D_1. For a BDD t and a domain D, we define the function $\mathsf{subtrees}_D$ to be defined as

$$
\begin{array}{lcl}
\mathsf{subtrees}_D(\mathit{false}) & = & \mathit{false} \\
\mathsf{subtrees}_D(\mathit{true}) & = & \mathit{true} \\
\mathsf{subtrees}_D((v, t_l, t_r)) & = & (v, t_l, t_r), \text{ if } v \notin D \\
\mathsf{subtrees}_D((v, t_l, t_r)) & = & \mathsf{subtrees}_D(t_l) \ \cup \ \mathsf{subtrees}_D(t_r), \text{ if } v \in D
\end{array}
$$

Claim 7.3 *Let $R = ((D_1, D_2), t)$ be a typed binary relation such that all variables in D_2 is higher than all variables in D_1. Then $(D_2, \mathsf{subtrees}_{D_1}(t))$ denotes the set of images $\{R(x) \ : \ x \in \{0,1\}^{|D_1|}\}$.* □

When we have the images, we can also compute the ranges for these images by looking at the path that leads to the subtree representing the image. For two BDDs t, t_{im} and a domain D wee define $\mathsf{paths}_D(t, t_{im})$ as follows:

$$\begin{array}{lcl} \mathsf{paths}_D((v, t_l, t_r), t_{im}) & = & (v, \mathsf{paths}_D(t_l, t_{im}), \mathsf{paths}_D(t_r, t_{im})), \text{ if } v \in D \\ \mathsf{paths}_D(t, t_{im}) & = & \mathit{true}, \text{ if } v \notin D \ \wedge \ t = t_{im} \\ \mathsf{paths}_D(t, t_{im}) & = & \mathit{false}, \text{ if } v \notin D \ \wedge \ t \neq t_{im} \end{array}$$

Claim 7.4 *Let $R = ((D_1, D_2), t)$ be a typed binary relation such that all variables in D_2 is higher than all variables in D_1, and let $I = (D_2, t_{im})$ be a typed relation denoting the image $R(b)$ under some bitstring $b \in \{0,1\}^{|D_1|}$. Then $(D_1, \mathsf{paths}_{D_1}(t, t_{im}))$ denotes the set of bitstrings $b' \in \{0,1\}^{|D_1|}$ such that $R(b') = R(b)$.* □

In the next section, we show examples of how this operation can be used to implement some of the operations on automata efficiently.

7.2 AUTOMATA

The framework of relations based on BDDs described in the previous section gives a basis for implementing a generic automata package.

It is important to stress that we are considering *nondeterministic* automata. There are other representations more specialized to deterministic automata, but we want to use nondeterministic automata to investigate the possible advantages such a representation could give us.

Let us recall the definition of automata. A finite automaton A over an alphabet Σ is a tuple (Q, S, Δ, F) where Q is a finite set of states, $S \subseteq Q$ is a finite set of *initial states*, $\Delta : Q \times \Sigma \times Q$ is a *transition relation*, and $F \subseteq Q$ is a finite set of *accepting states*.

All of these sets and relations will be represented using the BDD based relation representation presented in the previous section. Thus, the set of states is a set of bitstrings, and the transition relation is a relation between bitstrings. Even the alphabet is considered to be a set of bitstringsXFEXP .

Product Let us illustrate this representation by looking at the product of two automata. The two automata $A_1 = (Q_1, S_1, \Delta_1, F_1)$ and $A_2 = (Q_2, S_2, \Delta_2, F_2)$ have their transition relations Δ_1 and Δ_2 represented by a *typed relation* $((D_1^s, D_1^a, D_1^t), t_1)$ and $((D_2^s, D_2^a, D_2^t), t_2)$, respectively.

A product of A_1 and A_2 involves reasoning about tuples $(q_1, q_2) \in Q_1 \times Q_2$. The resulting automaton has states from $Q_1 \times Q_2$ where the transition relation Δ is defined by $\Delta((q_1, q_2), a, (q_1', q_2'))$ iff $\Delta_1(q_1, a, q_1')$ and $\Delta_2(q_2, a, q_2')$.

In terms of the typed relations, we need to adjust their domains such that we can take the product of the two BDDs t_1 and t_2 and get the result we want. First of all, we like the second component, the alphabet, to be the same. This is accomplished by assuring that $D_1^a = D_2^a$, and can be done by renaming the variables in one of the BDDs. Second, we want the domains of the first and third component two be disjoint in the two relations, i.e., such that $D_s^1 \cap D_s^2 = \emptyset$ and $D_t^1 \cap D_t^2 = \emptyset$. If we have these conditions, the typed relation $((D_s^1 \cup D_s^2, D_a^1, D_t^1 \cup D_t^2), t_1 \wedge t_2)$ denotes the desired relation Δ.

Minimization We will describe the minimization operation and how it takes advantage of the operation for finding images described in the previous section. The minimization algorithm involves finding a bisimulation relation $\sim$ that partitions the set of states Q. The algorithm finds this partitioning by iteration starting from a partition containing the final set of states F and the non-accepting set of states $Q \setminus F$.

In each iteration of the algorithm, a new partition is constructed from the old one. A function f from Q to sets of bitstrings is maintained as a representation of the current partition. The exact value of $f(q)$ for a state $q \in Q$ is not important, but whether for two states $q, q' \in Q$ we have that $f(q) = f(q')$ in which we say that the two states are in the same partition. To compute the new partition, we enumerate the current partition using a function $e : Q \mapsto \mathcal{N}$ such that $e(q) = e(q')$ iff q and q' belongs to the same partition . The function f' representing the new partition is then defined as follows:

$$f'(q) = \{(a, n, n') \ : \ \exists q' : \Delta(q, a, q') \ \wedge \ e(q) = n \ \wedge \ e(q') = n'\}$$

and is found by forming a relation of arity four and finding the images, using the technique described in the previous section. Each such image represents a part in the new partition, and using the technique for finding ranges, also described in the previous section, we find the set of states in each part.

7.3 EXPERIMENTAL RESULTS

The algorithms described in Chapter 2 have been verified with the execution times shown in the table below. The exact figures is not that important, other than that they are within a range that is reasonable for a verification tool.

Algorithm	Execution time in seconds
Bakery	18.12
Ticket	13.74
Szymanski	102.97
Dijkstra	203.79
Burns	32.79
Token Array (LIVENESS)	90.70
Termination Detection	44.99
Alternating Bit	79.77
Sliding Window	302.88

A note on the modeling of the sliding window protocol. Since this protocol contains both integer variables for the sequence numbers and an unbounded queue, we had to limit one of them since there is only one "dimension" of the words. The length of the queue was made bounded while the three integer variables were left unbounded, as was the sequence numbers in the queue. The limit of the queue length can of course be changed, but this will have effect on the execution time.

The execution time for the token array example is shown for verification of a liveness property: that for all processes we have that it eventually gets the token. Liveness properties take in general much more time to verify than safety properties, as expected. Safety properties for the token ring example can be verified in matters of seconds.

Chapter 8

Conclusions

In this thesis we have described a framework in which it is possible to describe several different types of infinite-state systems while still being able to perform automated verification. There are more specialized techniques for each of the types of infinite-state systems we have considered, and it may well be that these techniques are more efficient than ours, but our framework allows us to formulate these systems using a unified technique. The current implementation shows that the framework is not only theoretical, but is possible to implement with reasonable efficiency.

Regular model checking impose the restriction that the set of states and the transition relation must be regular. In the example of the sliding window protocol, the length of the queue had to be restricted to be able to represent both the queue and sequence numbers. This is because a word has only one dimension, and regular sets can only represent constraints between symbols that have a bounded "distance", since automata recognizing regular sets only have finite memory. Some tricks are sometimes necessary for the encodings such that the set of states and the transition relations have these properties. It remains to be seen to what extent these tricks can be automated.

The encoding into a regular model is now done manually. In the future, one can think of standard schemes of translating models into a regular model, having nice properties in terms of making the sets of states regular. For example, we have seen that some encodings of integer variables are better than others. This could be a part of a tool such that the algorithms may be specified on a higher level using e.g. integer variables and queues, and then being transformed automatically into a regular model.

The acceleration techniques presented in this thesis are able to emulate the acceleration operations for FIFO channels reported in Boigelot and Godefroind [BG96], but not those of Bouajjani and Habermehl [BH97] which also considers transitive closures that result in non-regular relations between words. Often when sets of states become non-regular it is because there are some linear constraints between

the number of occurrences of some symbol. An interesting direction is to combine regular sets with linear constraints, as considered by [BH97].

More work remains to make these techniques more efficient. This can be done by finding new ways to represent and operate on automata, but also by finding new ways of handling infinite compositions of regular relations.

Bibliography

ABJ98. Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, 1998.

ABJN99. Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

AD94. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

BCMD92. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

BEM97. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *Proc. Intern. Conf. on Concurrency Theory (CONCUR'97)*. LNCS 1243, 1997.

BG96. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

BGWW97. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, 1997.

BH97. A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. In *Proc. ICALP '97*, volume 1256 of *Lecture Notes in Computer Science*, 1997.

BJNT00. Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 2000.

Boi98. B. Boigelot. Symbolic methods for exploring innite state spaces, 1998.

Bry86. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

BSW69. K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmissions over half duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

Buc62. J. Buchi. a decision method in restricted second-order arithmetic, 1962.

BW94. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc.* 6^{th} *Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer Verlag, 1994.

Cau92. Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, Nov. 1992.

CC77. P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc.* 4^{th} *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

CGJ95. E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Lee and Smolka, editors, *Proc. CONCUR '95,* 6^{th} *Int. Conf. on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407. Springer Verlag, 1995.

CGP99. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

CJ98. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*. LNCS 1427, 1998.

DFvG83. E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed somputations. *Information Processing Letters*, 16(5):217–219, 1983.

FO97. L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronical Notes in Theoretical Computer Science*. Elsevier Science Publishers, July 1997.

FWW97. A. Finkel, B. Willems, , and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In *Proc. Infinity'97, Electronic Notes in Theoretical Computer Science*, Bologna, Aug. 1997.

GZ98. E.P. Gribomont and G. Zenner. Automated verification of Szymanski's algorithm. In *Proc. TACAS '98,* 4^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 424–438, 1998.

HJJ+96. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS '95,* 1^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, 1996.

JN00. Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. TACAS '00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 2000. to appear.

KMM+97. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc.* 9^{th} *Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

KMMG97. P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: A flexible toolset for monadic second–order logic. In *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), Enschede (NL)*, volume 1217 of *Lecture Notes in Computer Science (LNCS)*, pages 183–202, Heidelberg, Germany, March 1997. Springer–Verlag.

Koz97. Dexter C. Kozen. *Automata and Computability.* Springer-Verlag, 1997.

Lam74. L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

LPS93. Nancy A. Lynch and Boaz Patt-Shamir. Distrbuted algorithms, lecture notes for 6.852, fall 1992. Technical Report MIT/LCS/RSS-20, MIT, Jan. 1993.

McM93. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

MMP+95. Oliver Matz, Axel Miller, Andreas Potthoff, Wolfgang Thomas, and Erich Valkema. Report on the Program AMoRE. Technical Report 9507, Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1995.

Pnu77. A. Pnueli. The temporal logic of programs. In *Proc.* 18^{th} *Annual Symp. Foundations of Computer Science*, pages 46–57. IEEE, 31 October–2 November 1977.

Sis97. A. Prasad Sistla. Parametrized verification of linear networks using automata as invariants. In O. Grumberg, editor, *Proc.* 9^{th} *Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 412–423, Haifa, Israel, 1997. Springer Verlag.

SOR93. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

Szy90. B. K. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology*, pages 110–117, Los Alamitos, CA, 1990. IEEE Computer Society Press.

Tan96. Andrew S. Tannenbaum. *Computer Networks*. Prentice-Hall, 1996.

VW86. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc.* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

WB98. Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

WB00. Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

Licentiate theses from the Department of Information Technology

2000-001 Katarina Boman: *Low-Angle Estimation: Models, Methods and Bounds*

2000-002 Susanne Remle: *Modeling and Parameter Estimation of the Diffusion Equation*

2000-003 Fredrik Larsson: *Efficient Implementation of Model-Checkers for Networks of Timed Automata*

2000-004 Anders Wall: *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*

2000-005 Fredrik Edelvik: *Finite Volume Solvers for the Maxwell Equations in Time Domain*

2000-006 Gustaf Naeser: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

2000-007 Magnus Larsson: *Applying Configuration Management Techniques to Component-Based Systems*

2000-008 Marcus Nilsson: *Regular Model Checking*

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden