Aalborg University

Master Thesis

Regular Model Checking and Verification of Cellular Automata

Authors:
Joakim Byg
Kenneth Yrke Jørgensen

{jokke,kyrke}@cs.aau.dk

Department of Computer Science
Dat 6
June 10, 2008

Department of Computer Science

Aalborg University

Dat6

TITLE

Regular Model Checking and Verification of Cellular Automata.

PROJECT PERIOD

1. February 2007 -
10. June 2008

PROJECT GROUP

Computer Science, d602a

GROUP MEMBERS

Joakim Byg

Kenneth Yrke Jørgensen

SUPERVISOR Jiří Srba

NUMBER OF COPIES 4

PAGES IN REPORT 92

PAGES IN APPENDIX 7

PAGES IN TOTAL 99

Abstract

Regular Model Checking is a method for verification of infinite-state and parametric systems via regular languages and finite-state transducers. The configurations of the system are modelled using regular languages and the transition relation of the system is modelled with a transducer. Regular model checking is used to verify questions of the type, can we reach or avoid a given configuration (safety), or does all computations from each configuration reach a given configuration (liveness).

Cellular Automata is a model for describing state systems with local communication. Cellular Automata are used e.g. to model election algorithms, artificial life and car traffic.

The contribution of this thesis is divided into two main areas. First we, show the basics of regular model checking, then prove three results concerning the expressiveness of regular model checking. Secondly, we deal with Cellular Automata, the relation between different types of Cellular Automata and how we can analyse the behaviour of Cellular Automata of arbitrary size with regular model checking.

We carry out a number of experiments to test our prototype implementation, that can automatically convert a Cellular Automaton into a regular model checking framework.

Thesis Summary

RMC is method for verification for parametric systems and infinite state system. It was intoduced in 1997-1998 by Kesten et al. [14] and Boigelot and Wolper [31], and in 2000 Bouajjani et. al.

The idea of RMC is that the state space of an infinite or parametric system, can be described by a regular language, and that the transition relation between configuration can be represented as a regular relation between words. Several articles have shown different application of RMC for verification of token passing protocols, queues and mutual exclusion algorithms. In RMC we ask questions like, for any possible infinite number of initial configuration, can we avoid a set of bad configuration. In general the RMC framework is Turing powerful, this is why developing semi-algorithms to efficiently compute the reachable configuration, is an important part of work regarding RMC.

This thesis deals with two main areas.

The first area is concerned with the basics of RMC and theoretical parts of RMC. It defines and explains the basic concept of RMC, including a running example and description of a specific verification technique for RMC. Further it provided proofs of theorems about the expressiveness of RMC. The motivation for this is to formally formulate these folklore results and to further investigate the properties of RMC. The report provides full proofs for the claims, that RMC is Turing powerful, that by considering only a finite set of initial configurations the verification problem becomes PSPACE-complete and lastly, that length preserving transducers can model non-length preserving ones.

The first result obviously states that RMC is by its nature undecidable. The second result is proved via a reduction from one-safe Petri nets, and so provides a sketch of how to verify bounded Petri nets with RMC. This result also illustrates that RMC can be used for bounded model checking. The last result concerning length and non-length preserving transducers preserve the possibility to verify safety properties, but verifying liveness properties remains equally problematic — however we briefly sketch how a class of non-

length preserving transducers can be verified for liveness properties.

Second part of the thesis deals with the class of computational devices called Cellular Automata (CA). A CA are state-evolution system with local communication, that are used for e.g. simulation of car traffic and simulation of artificial life. The CA contains a cellular space, which can be multi-dimensional. Each cell in the cellular space know of the states of the surrounding cells. At an update signal, all cells evolve simultaneously to a new state, based on their neighbours. The thesis focuses on the restricted CA types, where the cellular space is finite, called bounded CA (bCA), and where a finite cellular space is connected in a ring typology, called circular CA (cCA). Also these are defined to allow nondeterministic evolutions. The thesis shows a linear reduction from bCA to cCA, and an exponential reduction from cCA to bCA. Both reductions are up to isomorphism.

Finally the thesis contains a reduction from reachability in cCA to RMC. This reduction is used to show how RMC can be used for parametric verification of bCA and cCA. The results of the conducted experiments illustrate that some CA algorithms are hard to describe regularly, but also that rather complex behaviour is still verifiable in short time. The properties verified is either properties avoided or that there exists a configuration in a given CA that satisfies the given property. The experiments are conducted so that the number of cells in the initial configurations are parametrised. Which leaves the future work of verifying the problems so that a given property holds for every initial configuration.

Preface

This Master Thesis is written as a part of the specialisation year at the Department of Computer Science - Aalborg University. The thesis builds upon the work in our specialisation midterm report “Regular Model Checking” [8], developed at the first half of the specialisation year. Parts of the content of this thesis are reused from the midterm report [8]. Section 2.1 concerning the RMC definitions is reused with small correction and little rewriting. The proofs and the proof idea of of Theorems 3.1, 3.2 are reused with some correction and the proof for Theorem 3.3 is reused, but with significant corrections and small modifications. Section 2.3 concerning the related work is to some extend rewritten. We will not make any references to our self when reusing these parts.

Acknowledgement We would like to thank our supervisor Jiří Srba, for excellent supervision, and for always having his door open for question answering. We have very much explicated this. A thank also goes to Tomáš Vojnar, for providing us with the ARMC tool, and for feedback and comments on some of our ideas. Lastly we would like to thank our social association, the F-Klub[15]. Thanks for many great social events and experiences — and for providing us with vitals (coffee and cola).

Contents

Chapter 1

Introduction

Today many tasks are automated by computers — and of course we expect that these tasks are carried out correctly. Often this is a question of implementing given algorithms correctly. But it is also a matter of designing the algorithms correctly, so that they do what they are intended to do — without any exception. In parallel and/or possibly non-terminating systems it can be quite hard for designers to reason about the exact behaviour of the system, though this can be very vital. As an example, consider a safety critical systems like airbag release or flight traffic management systems. Correct behaviour of such systems is crucial, since malfunction might result in severe injuries or casualties.

Even though the designer should guarantee, on the basis of his/her analysis and design, that the system does not fail, some safety critical systems can not be subject to human failures. These systems need to be *proven* that they behave according to their specifications. This is where formal and automatic verification of systems comes into play. Formal verification proves that a system behaves according to some specified property. However, there is a complexity issue to take into account — the actual system can include a lot of irrelevant behaviour with regards to a given property. So checking the systems behaviour, line for line, is near impossible. Instead we can make some symbolic representation of the important parts of the system and then automatically verify properties of this representation. This concept is referred to as *Model Checking*, — given a model, we check if the model complies with a given specification.

In general we can model a system as a *Labelled Transition System* (LTS), which is a directed graph over the set of configurations (also called the state-space) that a given system can be in. The vertices of the LTS are the configurations and the edges are representing the transition relation over the set of configurations.

With a model of the system we can ask whether certain properties are satisfied or not, e.g. if we have a model of a mutex algorithm we would like to know if our model complies to the mutual exclusion requirement. Such a property is what we call a *safety* property. Safety properties often describe situations that we would like to avoid in our model. In the case of mutex we would like to avoid two or more process being in the critical section at the same time.

Sometimes we also want to make sure that no matter which configuration we consider we can always reach some configuration with a "good" property. With regards to the mutex example, it could be that we want to make sure that each processes can eventually get into the critical section. Properties like this are referred to as *liveness* properties.

The lowest level of a model is in general a LTS, and so it is often feasible to use some abstraction to describe the LTS. One such abstraction is the Calculus of Concurrent Systems (CCS)[17] developed for concurrent processes by R. Milner in 1980. This language makes use of a handshake concept so that different processes can communicate via handshake synchronisations. R. Milner et. al. later extended CCS so that the channels can process names. This is referred to as the π-calculus. Considering CCS, a model checking strategy would be to check if a given CCS model is equivalent, for some equivalence, to some specification. E.g. the equivalence could be weak bisimulation and a specification could be modelled also in CCS.

To check if a property is satisfied in the model we have to explore the state-space of the system, to see if one or more states violate or satisfy the property. However, we often have the problem that the state-space is quiet large, possibly infinite, making it hard or impossible to iterate all possible configuration in the state-space. This problem is often referred to as the state-space explosion problem.

Normally we divide systems into finite state system and infinite state systems. Most interesting systems has by nature an infinite state-space, just consider a program that contains a single integer variable. As an integer can be arbitrary large, we have a state-space with infinitely many states.

As a consequence of the problem regarding state-space explosion, an important part of model checking is finding an efficient representation of the state-space. For finite state-space systems a common way to represent the state-space is by using Binary Decision Diagrams (BDD).

Over the years tools have been developed to facilitate domain specific or just easy modelling and automatic verification. The website [32] shows a list of different types of verification tools, including tools for real-time systems.

In 1997-1998 Kesten et al. [14], Boigelot and Wolper [31] proposed using regular languages as representations for infinite state-space systems, and parametric systems. In 2000 Bouajjani et. al. presented *Regular Model Checking* (RMC) as a uniform framework for algorithmic verification of infinite state-systems[7]. The fundamentals of RMC, is to represent the possibly infinite state-space, as a finite-state automaton, and to represent the transition relation between the configuration as a regular relation. The task is now to calculate the reachable configurations efficiently.

Regular Model Checking To explain the basics of RMC we will use a simple token passing protocol as example. This protocol will be used through the report as a running example.

The token passing protocol works on a number of connected processes labelled 1 to n. The process can either hold the token (T), or not hold the token (N). A given process with the token can pass the token to its right neighbour.

We can now model this protocol using RMC. In RMC we use a word over the alphabet $\{T, N\}$ to describe the possible configuration of this system. Given a system with $n = 5$ processes, the word $NNTNN$ means that the process labelled 3 has the token, and that processes labelled 1, 2, 4 and 5 do not have the token.

We can describe the behaviour of the system using a relation. In this case the relation will be that words on the form (N^iTNN^j) will be rewritten to words of the form (N^iNTN^j) for any numbers $i, j \geq 0$. We can describe such a relation using a finite transducer.

The initial configuration of the system would be one where the first process holds the token. We can for all possible numbers of processes, describe this using the regular expression TN^*.

This shows how we can encode a problem using RMC, the task is now how to calculate the reachable configuration. In this example it is easy to see that for each configuration, the token is passed to the right neighbour, $TNNNN \rightarrow NTNNN \rightarrow \cdots$. Hence the set of all reachable configurations is expressed by the regular expression: N^*TN^*.

Contributions The contributions of the thesis are divided into two areas. The main focus of this thesis is in the area of regular model checking. So first we demonstrate results concerning the expressiveness of RMC. Second we show how we can apply RMC to perform verification of *Cellular Automata* (CA).

In the thesis we provide the proofs for the folklore result that in general RMC is undecidable. We prove that by restricting the initial configurations to be finite we get that RMC is PSPACE-complete. And lastly we show that it makes no difference when we consider safety properties if we use length or non-length preserving transducers. Together with these results, we also show how we can reduce reachability of two-counter Minsky machines and one-safe Petri nets to RMC.

In the second part of the thesis we focus on Cellular Automata (CA), in particular we focus on one dimensional non-deterministic bounded and circular CA, which are a subclass of the CA. We show how we can perform parametric analysis of circular CA, by giving a reduction from reachability analysis of circular CA to RMC. We also prove that circular and bounded CA are equal up to isomorphism. We provide a tool, for automatic conversion of CA into RMC, and carry out a number of experiments for different types of CA, to test the automatic reduction and the usefulness of RMC as verification of CA.

Thesis Outline In Chapter 2 we will formally define the area of RMC, and give an example of RMC. We also outline the work done in the area, and go into detailed of an acceleration method used for RMC. In Chapter 3, we present proof of three results concerning the expressiveness of RMC. We also give a short discussion of how we can preserve liveness checking for some special non-length preserving transducers. Chapter 4 introduces and defines the model of Cellular Automaton. Chapter 5 we present the result for isomorphism between circular and bounded CA and bounded and circular CA. This chapter also shows how we can reduce circular and bounded CA to RMC. In Chapter 6 we present the result of our experiments of verification of some interesting CA with the use of our reduction tool and the ARMC tool. The thesis is concluded in Chapter 7.

Chapter 2

Regular Model Checking

In this chapter we introduce regular model checking (RMC) formally. First we will introduce a number of definitions concerning RMC. These will be used throughout the report. Hereafter we will give a definition of RMC as a decision problem, followed by an example of RMC, and lastly we describe the acceleration technique Abstract Regular Model Checking. The tool we have used for our experiments is an implementation of this technique.

2.1 Definitions

In this section we introduce a number of definitions, which we use throughout the following chapters.

We denote the set of natural numbers $\{1, 2, \ldots\}$ by $\mathbb{N}$, and let $\mathbb{N}_0$ denote the set of non-negative integers. The set $\mathbb{Z}$ is the set of positive and negative integers.

We say that a finite set of symbols is an alphabet and it by Σ. The set of all strings over Σ is denoted Σ^*. The identity element of Σ^* is the empty string, denoted ε. A set of strings over the alphabet Σ is called a language. We denote the length of the string w as $|w|$.

Finite Automaton

Definition 2.1. A *nondeterministic finite automaton* (NFA) is a 5-tuple, $M = (Q, \Sigma, q_0, F, \delta)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $q_0 \in Q$ is the start state,

- $F \subseteq Q$ is the set of accept states, and
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \longrightarrow 2^Q$ is the transition function.

$\diamondsuit$

We say that a finite automaton $M = (Q, \Sigma, q_0, F, \delta)$ *accepts* a string $s = \alpha_1\alpha_2 \cdots \alpha_n$, where $\alpha_i \in \Sigma \cup \{\varepsilon\}$ for all i, $1 \leq i \leq n$, if there is a sequence of states, $q_0, q_1, \ldots, q_n$, where it holds that:

- q_0 is the start state,
- $q_{i+1} \in \delta(q_i, \alpha_{i+1})$ for $i = 0, \ldots, n-1$, and
- $q_n \in F$.

The set of all strings that M accepts is the language *recognised* by M and is denoted $L(M)$.

We say that a language L is a *regular language* if L is recognised by a finite automaton.

Theorem 2.2 ([24])**.** *Regular languages are closed under* $\cup$*,* $\cap$*,* $\circ$*,* $-$ *and* **.*

Deterministic Finite Automaton

Definition 2.3. A *deterministic finite automaton* is a nondeterministic finite automaton as defined in Definition 2.1, but where $|\delta(q, a)| \leq 1$ and $|\delta(q, \varepsilon)| = 0$ for all $q \in Q$ and $a \in \Sigma$. $\diamondsuit$

Theorem 2.4 ([24])**.** *Every nondeterministic finite automaton M_n has an equivalent deterministic finite automaton M_d, where $L(M_n) = L(M_d)$.*

Finite-state transducer

Definition 2.5. A *finite-state transducer* is a 5-tuple $T = (Q, \Sigma, q_0, F, \delta)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states, and
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \longrightarrow 2^Q$ is the transition function.

◊

We use $q \xrightarrow{\alpha/\alpha'}_{\delta} q'$, or just $q \xrightarrow{\alpha/\alpha'} q'$ if δ is clear from the context, if $q' \in \delta(q, (\alpha, \alpha'))$.

We say that a finite-state transducer $T = (Q, \Sigma, q_0, F, \delta)$ *translates* string $s = \alpha_1\alpha_2 \cdots \alpha_n$ to string $s' = \alpha'_1\alpha'_2 \cdots \alpha'_n$, where $\alpha_i, \alpha'_i \in \Sigma \cup \{\varepsilon\}$ for $1 \leq i \leq n$, if there exists a sequence, $q_0 \xrightarrow{\alpha_1/\alpha'_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n/\alpha'_n} q_n$, where it holds that

- q_0 is the start state,
- $q_i \xrightarrow{\alpha_{i+1}/\alpha'_{i+1}} q_{i+1}$ for $i = 0, \ldots, n-1$, and
- $q_n \in F$.

We call such a sequence a *translating computation* of the finite-state transducer T.

A finite-state transducer T defines a regular relation $[T]$ over Σ^*.

Given a finite-state transducer T and strings $s, s' \in \Sigma^*$. We have $(s, s') \in [T]$ if T has a translating computation that translates from s to s'. We call s the *input string* and s' is the *output string.*

Let $T = (Q, q_0, \Sigma, F, \delta)$ be a finite-state transducer, where $q \in Q$, $a, b \in \Sigma$, $w_1, w_2 \in \Sigma^*$ and string $s = w_1 a w_2$, then $q \xrightarrow{a/b} q'$ intuitively means that when T is in the state q and reads a symbol a from a string s, then a is exchanged with b and T moves to the state q'.

Example 2.6. Let $T = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \{q_3\}, \{q_0 \xrightarrow{a/a} q_0, q_0 \xrightarrow{\varepsilon/b} q_1, q_1 \xrightarrow{a/\varepsilon} q_2, q_0 \xrightarrow{\varepsilon/\varepsilon} q_2, q_2 \xrightarrow{a/a} q_2, q_2 \xrightarrow{\varepsilon/a} q_3\})$ be a finite-state transducer. We shall often use only the graphical representation to define transducers, as in Figure 2.1.

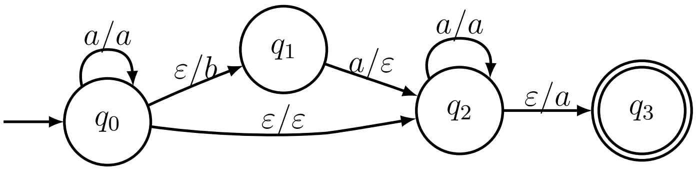


Figure 2.1: Graphical representation of transducer T

The graphical representation is a directed graph, where the states are the vertices, δ is the connecting edges between the states. The start state is pointed to by an edge with no starting state. The accept states are highlighted by a double lined circle. The first symbol on a given edge is the input symbol and the second symbol is the output symbol.

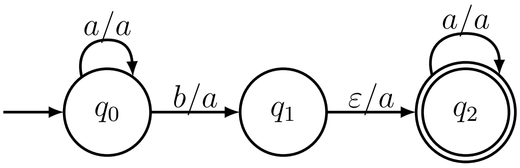
Now $(aaa, aaba) \in [T]$, as the following translating computation exists: $q_0 \xrightarrow{a/a} q_0 \xrightarrow{a/a} q_0 \xrightarrow{\varepsilon/b} q_1 \xrightarrow{a/\varepsilon} q_2 \xrightarrow{\varepsilon/a} q_3$, where q_3 is an accepting state. △

Deterministic finite-state transducer

Definition 2.7. A *deterministic finite-state transducer* is a finite-state transducer as defined in Definition 2.5, but where for every $q \in Q$, $a, b \in \Sigma$ and $\alpha \in \Sigma \cup \{\varepsilon\}$, either $|\delta(q, a, b)| \leq 1$ and $|\delta(q, \varepsilon, \alpha)| = 0$ or $|\delta(q, \varepsilon, \alpha)| \leq 1$ and $|\delta(q, a, b)| = 0$.

◊

Example 2.8. Let T be a finite-state transducer as follows:



Then T is an example of a deterministic finite-state transducer. △

Length-preserving Transducer

Definition 2.9. A transducer $T = (Q, \Sigma, q_0, F, \delta)$ is called *length preserving* if it satisfies that there are no transitions on the forms $q \xrightarrow{\varepsilon/a} q'$ or $q \xrightarrow{a/\varepsilon} q'$, where $a \in \Sigma$. ◊

Observe that for all pair of strings $(s, s') \in [T]$ it then holds that $|s| = |s'|$.

Example 2.10. As an example of a length-preserving transducer see Figure 2.2.

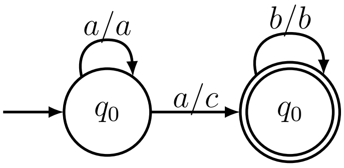


Figure 2.2: Example of a length preserving transducer.

Where as the transducer in Figure 2.1 is non-length preserving, as we see that it has rules on the form $q \xrightarrow{\varepsilon/a} q'$ and $q \xrightarrow{a/\varepsilon} q'$. △

Union of Transducers

Definition 2.11. Let $T_1 = (Q_1, \Sigma_1, q_{0_1}, F_1, \delta_1)$ and $T_2 = (Q_2, \Sigma_2, q_{0_2}, F_2, \delta_2)$ be finite-state transducers, where $Q_1 \cap Q_2 = \emptyset$. The *union* of T_1 and T_2 is defined as $T_1 \cup T_2 = (Q, \Sigma, q_0, F, \delta)$, where $q_0 \notin Q_1 \cup Q_2$ and

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$
- $F = F_1 \cup F_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta(q, \alpha, \alpha') = \begin{cases} \delta_1(q, \alpha, \alpha') & \text{if} \quad q \in Q_1 \\ \delta_2(q, \alpha, \alpha') & \text{if} \quad q \in Q_2 \\ \{q_{0_1}, q_{0_2}\} & \text{if} \quad q = q_0 \text{ and } \alpha = \alpha' = \varepsilon \\ \emptyset & \quad\text{otherwise} \end{cases}$

So $(s, s') \in [T_1 \cup T_2]$ iff $(s, s') \in [T_1]$ or $(s, s') \in [T_2]$. ◊

Example 2.12. Two transducers T_1, T_2 and the union of these $T_1 \cup T_2$, are illustrated in Figure 2.3.

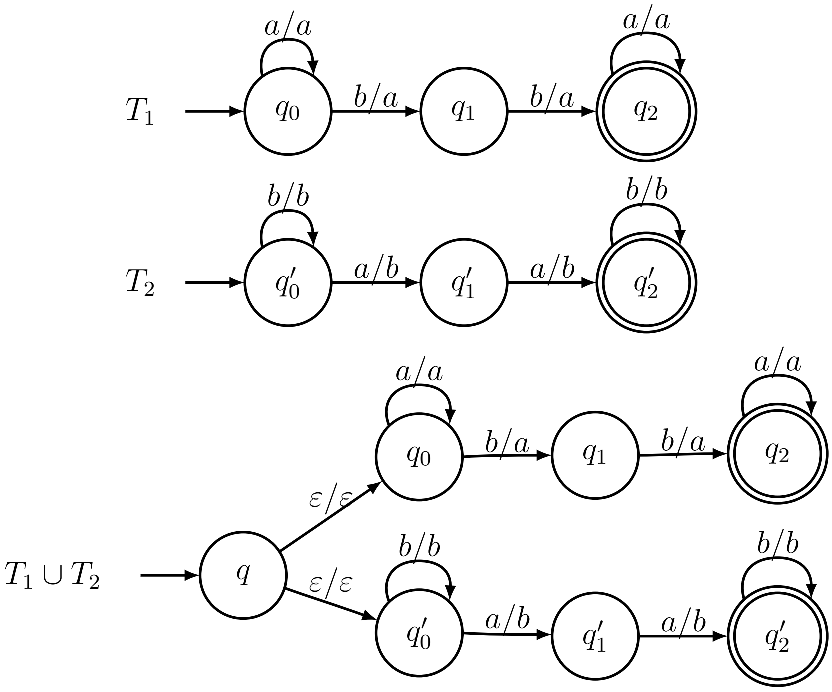


Figure 2.3: Two transducers and the union of these.

The union is made by adding a new start state q, and adding ε/ε transitions from q to the start states in T_1 and T_2. This new transducer, $T_1 \cup T_2$, nondeterministically guesses whether the string is accepted by T_1 or T_2. $\triangle$

Composition of Transducers

Definition 2.13. Let $T_1 = (Q_1, q_{0_1}, F_1, \Sigma_1, \delta_1)$ and $T_2 = (Q_2, q_{0_2}, F_2, \Sigma_2, \delta_2)$ be finite-state transducers. We say that $T_1 \circ T_2 = (Q, q_0, F, \Sigma, \delta)$ is the *composition* of T_1 and T_2, where

- $Q = Q_1 \times Q_2$
- $q_0 = (q_{0_1}, q_{0_2})$
- $F = F_1 \times F_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $(q_1', q_2') \in \delta((q_1, q_2), (\alpha, \alpha'))$ if for some $\alpha'' \in \Sigma \cup \{\varepsilon\}$ there exists $q_1' \in \delta_1(q_1, (\alpha, \alpha''))$ and $q_2' \in \delta_2(q_2, (\alpha'', \alpha'))$ where $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$ and $\alpha, \alpha', \alpha'' \in \Sigma \cup \{\varepsilon\}$.

Note that even if T_1 and T_2 are deterministic, $T_1 \circ T_2$ can be nondeterministic.

Let T_1, T_2 be finite-state transducers, then finite-state transducer $T_1 \circ T_2$ defines a relation $[T_1 \circ T_2]$ where $(s, s') \in [T_1 \circ T_2]$ if there exists a string s'' such that $(s, s'') \in [T_1]$ and $(s'', s') \in [T_2]$.

Let S be a set of strings and T a finite-state transducer. By abuse of notation, we let $S \circ T$ denote the set of strings S', where for all strings $s' \in S'$, we have $(s, s') \in [T]$, where $s \in S$. If S is regular, then by Definition 2.1 there is a finite automaton that recognises S. In general we do not distinguish between finite automata and regular sets with regards to composition of regular sets and transducers.

We denote the transducer $T \circ T$ as T^2, and in general $T^n = T^{n-1} \circ T$. We define $T^* = \bigcup_{n \in \mathbb{N}_0} T^n$, as the reflexive, transitive closure of T.

$\diamondsuit$

Example 2.14. Recall the token passing example from the introduction. The relation defined in that example can be expressed by the transducer T in Figure 2.4.

Given the transducer T we compose $T \circ T$ or T^2, illustrated in Figure 2.4.

The set of states in T^2 is easily obtained by $Q \times Q$. Now we add transitions by imagining that we have two transducers, T_1 and T_2, equal to T. We have

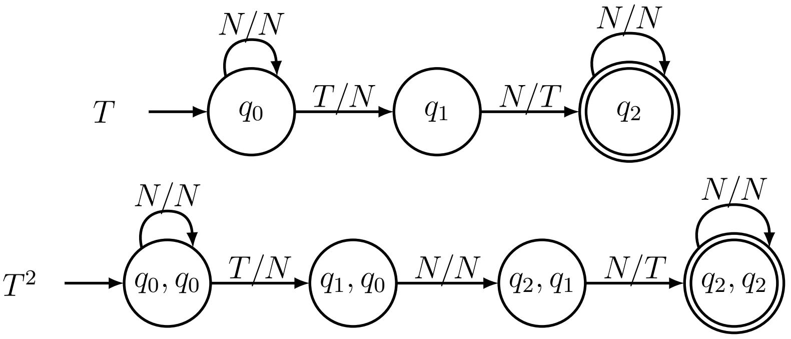


Figure 2.4: Transducer T simulates one step of a token passing protocol. The composition, T^2, simulates two steps of the protocol.

to do an exhaustive analysis of all edges in T_1 and T_2 to create the transition function δ of T^2.

$\triangle$

2.1.1 Regular Model Checking

Regular Model Checking (RMC) is a method for automatic verification of transition systems. Verification is done using a regular model checking framework.

When verifying a system we create a RMC-framework consisting of:

- A set of strings describing the initial configurations for the system.
- A set of strings describing the "bad" states that we want to avoid.
- A finite-state transducer describing the transition relation between configurations of the system.

A configuration[1] is a snapshot of the transition system. The transition relation describing the progress of the system from a configuration to a new configuration.

Definition 2.15. A RMC-framework is a 3-tuble $R = (I, T, B)$ where

- I is a regular set,
- T is a finite-state transducer, and

[1] Configurations are often also refered to as states, we use the term configuration to avoid confusion with the state of finite-state automata and transducers.

- B is a regular set.

$\Diamond$

We can formulate RMC as a decision problem. In most literature RMC is defined as the problem

Definition 2.16. Decision Problem: *RMC*

Instance: An RMC-framework $R = (I, T, B)$, where T is length preserving.

Question: Is $(I \circ T^*) \cap B = \emptyset$?

$\Diamond$

Even though most literature focuses on length preserving transducers, we still have the more general instance of the problem, where the transducer is allowed to be non-length preserving. We formulate this as the decision problem *GRMC*:

Definition 2.17. Decision Problem: *GRMC*

Instance: An RMC-framework $R = (I, T, B)$, where T is allowed to be non-length preserving.

Question: Is $(I \circ T^*) \cap B = \emptyset$?

$\Diamond$

Finally we define the decision problem *RMC-finite*. Here the initial set is finite and the transducer is length preserving. This problem stands out because we only have finitely many reachable configurations.

Definition 2.18. Decision Problem: *RMC-finite*

Instance: An RMC-framework $R = (I, T, B)$, where T is length preserving and I is finite.

Question: Is $(I \circ T^*) \cap B = \emptyset$?

$\Diamond$

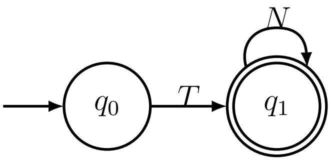
2.2 Example of Regular Model Checking

In this section we present a detailed example of how RMC works.

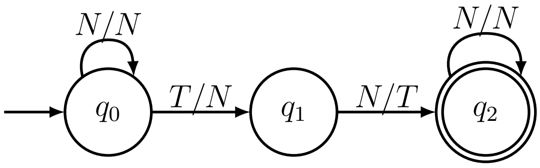
As an example we will use the token passing protocol as described in the Introduction on 11.

We create a RMC-framework as defined in Definition 2.16, $R = (I, T, B)$, where T is length preserving.

In the case of the token passing protocol we can describe the initial (parameterised) set by the automaton $M = (Q, \Sigma, q_0, F, \delta)$:

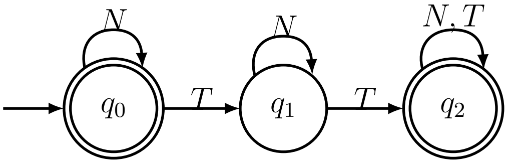


We have previously seen that the transition relation of the protocol can be described by a relation. This is a regular relation and so it can be described by a length-preserving transducer $T = (Q, \Sigma, q_0, F, \delta)$:



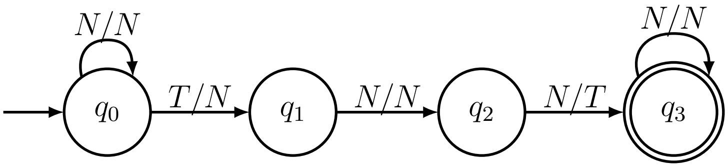
By a view of this transducer we can see that it moves the token one place to the right.

We now have a RMC encoding of the transition system. To do the verification we need to express a property we wish to check for. In this case we would like to check for the property: “there is always exactly one token?”. We can express this property using a regular language. In RMC we reformulate this to a “bad” property that our model should avoid, that is “there are more or less than one token?”. The bad set is as follows:

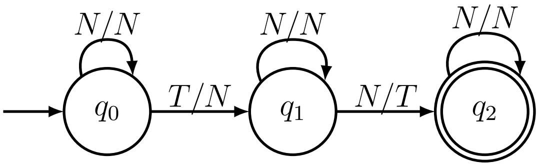


After building the RMC-framework, we can now perform the verification. The task now consists of calculating the reachable configurations and see if there is an intersection with the bad set. We can calculate the reachable configuration by repeatedly applying the transducer to the initial set. In this example by applying the transducer once to the initial set, we get the languages described by the regular expression NTN^*. And by applying the transducer again we get language described by $NNTN^*$. Observe that this approach is clearly never going to end and so we will not be able to say anything about the system.

Instead of applying the transducer repeatedly, we can create a transducer that has the effect of applying the transducer several times. To do this we can calculate the composition of T with it self. Recall Definition 2.13 on page 18 of composition of transducers, then the transducer $T \circ T$ is the following transducer:



By applying this transducer to I we get the reachable configurations $NNTN^*$, as expected. If we can calculate the reflexive, transitive closure of T, denoted T^* then we can calculate all reachable configurations by applying this to the initial set. The problem is how to calculate T^*. Here we will need an acceleration technique to speed up the computation. In Section 2.3 we shortly introduce a number of different techniques for accelerating this calculation. However in this simple example, it is fairly easy to see that the transducer for T^* will be:



By applying T^* to I, we see that the set of reachable configurations is N^*TN^*. Now by calculating the intersection with B, we can check if some of the configurations in B are included also in $I \circ T^*$. In other words we check if $I \circ T^* \cap B \overset{?}{=} \emptyset$. In the case where the intersection is empty, we know that the "bad" configurations are avoided.

2.3 Related Work

In this section we will look at the work already done in the area of RMC. As mentioned in general RMC is undecidable, that is way an important part of

RMC is to develop good simi-algoirhms for calculating the effect of applying the transducer an arbitrary number of times on the initial set.

The article ”A Survey of Regular Model Checking” [4] from 2004 summarises the work in the area of RMC. The article divides the approaches into tree overall categories: quotienting, abstraction and extrapolation.

Quotienting The idea is to find a suitable equivalence relation of the states in the transducer, and by quotienting the transducer finding a approximation of the transducer composed with it self.

In general quotienting increases the relation defined by the transducer. This problem has been solved by introcucing special constraints to the equivalence relation. The work done in [2, 7, 10, 1] all uses a quotienting technique to find the transitive closure of transducers.

Extrapolation In the extrapolation approaches, the transitive closure of the transducer is calculated by calculating the effect of composing the transducer a number of times with it self, and the guessing the growth pattern.

In [7, 25] a method for extrapolation is presented. They also show that under some special conditions the extrapolation is an exact limit.

In [5] Boigelot et. al. present a technique for Presburger arithmetic expressions, with a binary encoding of numbers. They extend the extrapolation technique by using sample points, a trick that utilise that the transducer is reflexive. Then we only need to consider a number of fix points, when looking at the transducer. They also present an efficient dominance relation to speed up the determination procedure of transducers.

Habermehl and Vojnar [13] uses a different extrapolation approach. Their approach calculates the exact set of reachable configuration up to some length n. Now if some configuration violates the property, we have an counterexample, else the set of configuration of arbitrary length is extrapolated. If the extrapolated set does not violate the property, the property is not violated, else the n is increased and the computations are done again.

As the extrapolation is a over approximation we might encounter spurious counter-examples, that must be handled and detected.

Abstraction In the computation of the transitive closure of the transducer, there might be parts that are computably large and complicated, and might even unused or irrelevant for it the property holds. By applying the abstract-check-refine paradigm into the computation of the closure, we might accelerate the computation, so instead of calculating the reachable configuration, a over approximation is calculated.

Often a over-approximation will not give a negative answer, but a maybe answers, and we then need to refine the abstraction, to conclude, if a positive answer is given we know that the property is satisfied.

When abstracting and refining, it might be the case that we continue refining infinitely. Often, when this is an equivalence relation with a finite index, termination is guaranteed and the abstraction converges to a (exact) limit.

The article [6], by Bouajjani et. al. describes four ways to decide candidates for state merging, along with a way to detect if a ”maybe” answer is indeed correct. In Section 2.4 we will have a closer look at this method.

Other work The work done in [3] propose to use the Linear Temporal Logic, with monadic second-order locig (LTL(MSO)) to specify transducers, sets and validation properties of a given model. This makes it easier to specify a fairness requirement in the model.

It seems that in resent years there is devoted more time to the area of tree model checking, where tree languages is used in the same way as a regular languages in RMC.

Several articles tests there acceleration techniques using a prototype implementation of their method. However after searching for these, we found only very few tools thatsvn are published, and those which was, where no longer under development, and several of them where no longer accessible. The only public accessible RMC tool we where able to get, was that of `regularmodelchecking.com` [22].

2.4 Abstract Regular Model Checking

For our experiments and implementation we have used the ARMC prototype tool developed by A. Bouajjani et. al.[6]. It was kindly lent to us by T. Vojnar. The ARMC tools implements the acceleration technique used in the article Abstract Regular Model Checking [6]. In this section we will look at how this acceleration techniques works.

Bouajjani et. al. state in the article that it is often enough to calculate an over approximation of the set $I \circ T^*$ to get answers for verification question $I \circ T^* \cap B \overset{?}{=} \emptyset$. They computes the over approximation of $I \circ T^*$ by iteratively computing over approximations as illustrated in Figure 2.5. The computation is guaranteed to halt as long as the abstraction function α, is finitary. In practise this means that the computation will reach a fix point, where the last iterative overapproximation is in fact the same as the set it was calculated from. Later in this section we give a couple of examples of how abstraction is done.

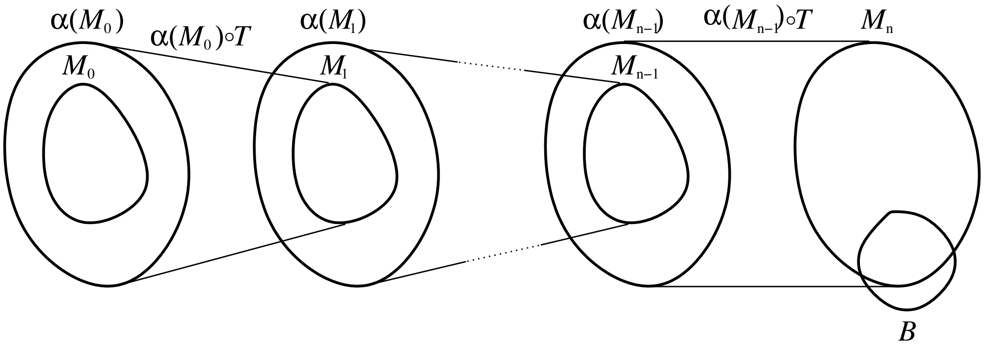


Figure 2.5: Iterative overapproximation, where M_i is a finite automaton representing a set of strings and α is the abstraction function.

In Figure 2.5 we see a example of the approximation. The resulting overapproximation is intersecting with the set B. If the computation ends with an intersection like this we will have to find out if this is indeed an intersection with the actual set $I \circ T^*$ and not just the over approximation.

This is done by applying the inverse transducer T^{-1} to the intersection and then checking if the resulting set is intersecting with the previous abstraction, if this is the case, then T^{-1} is applied to this new intersection as is illustrated in Figure 2.6. This procedure is repeated until we either reach the initial set and we have found a real intersection or we at some point while backtracking encounter an empty intersection, in which case we found a spurious counterexample and will have to refine the abstraction function.

One way is to refine α to α', where for every set that is disjoint from the encountered intersection, will not be approximated to a set that is not disjoint from the intersection. The other way is to simply exclude the encountered intersection from the last abstraction and then continue to do iterative abstractions as before — even though the intersection set might ”sneak” into the abstraction again.

This is the overall approach used in abstract regular model checking. Now let us have a look into how abstractions are done.

Abstraction is done locally on the automata representation of a given set of configurations $M = (Q, \Sigma, q_0, F, \delta)$. The states of the automaton M are compared to an finite automaton P defining a predicate language.

There are two ways they can be compared and merged: (1) Two states $q_1, q_2 \in Q$ are merged if they have the same forward and/or backward behaviour for up to some given length. (2) Two states $q_1, q_2 \in Q$ are merged if

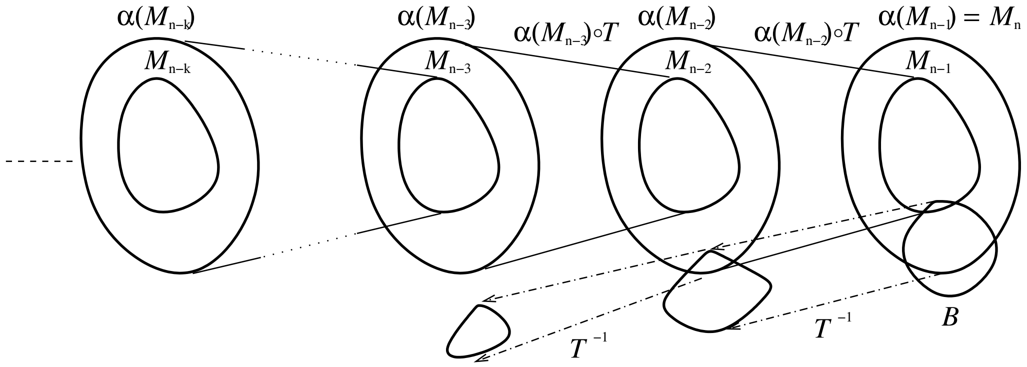


Figure 2.6: Backtracking the intersection with the over approximations. In this case we illustrate that we have found a spurious counter example.

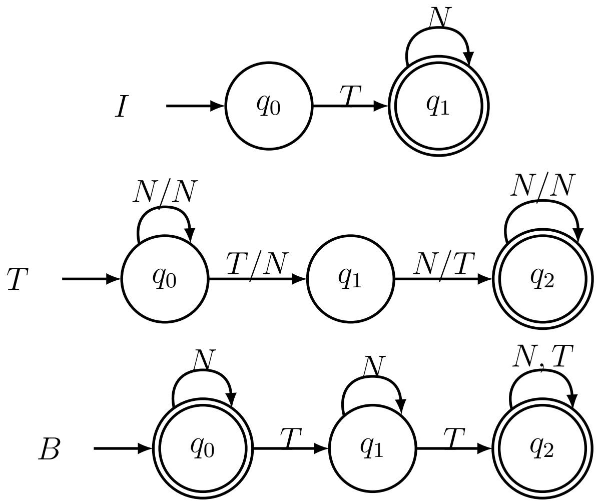
both their forward and/or backward predicate language is intersecting with the language of the predicate automaton $L(P)$.

1. Two states are merged if they share the same behaviour for a predefined number of transitions. This means that if from two states there exists two equivalent traces of the given length, they are merged. This is possible for both forward and/or backwards traces.

2. Given finite automaton $M = (Q, \Sigma, q_0, F, \delta)$, we say that the forward language of the state $q \in Q$ is the set:
$$L(M, q) = \left\{ a_0 a_1 a_2 \cdots a_n \;\middle|\; \begin{array}{l} a_i \in \Sigma, \text{ and there is a computation} \\ q \xrightarrow{a_0} q' \xrightarrow{a_1} q'' \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_F, \text{where } q_F \in F \end{array} \right\},$$
and similarly we say that the backward language of the state q is the set:
$$\overleftarrow{L}(M, q) = \left\{ a_0 a_1 a_2 \cdots a_n \;\middle|\; \begin{array}{l} a_i \in \Sigma, \text{ and there is a computation} \\ q_0 \xrightarrow{a_0} q' \xrightarrow{a_1} q'' \xrightarrow{a_2} \cdots \xrightarrow{a_n} q \end{array} \right\}.$$

   Now abstraction by forward and backward predicate language respectively of a given automaton $M = (Q, \Sigma, q_0, F, \delta)$ and the states $q_1, q_2 \in Q$ by means of predicate automaton $P = (Q', \Sigma', q'_0, F', \delta')$, is done by detecting if the predicate language $L(P)$, is intersecting with either $L(M, q_1)$ and $L(M, q_2)$ or $\overleftarrow{L}(M, q_1)$ and $\overleftarrow{L}(M, q_2)$. If both $L(M, q_1)$ and $L(M, q_2)$ have nonempty intersection with $L(P)$, then the states q_1 and q_2 are merged. Likewise with regards to the backwards language.

Example 2.19. Recall Section 2.2, where we went through the token passing protocol encoded as the following RMC-framework $R = (I, T, B)$:



Now let us merge some states by the two abstraction methods. First we illustrate an example of abstraction by bounded forward trace of states, then we show how to do abstraction by predicate languages and then we how backtracking and refinement is done.

Now have a look at automaton C_3 in Figure 2.7. E.g. the behaviour of q_0 and q_3 have the same forward trace up to length two. The automaton $\alpha_2^F(C_3)$ in Figure 2.7 illustrates the result of merging the states that have equivalent forward traces up to length two. If the length of the considered traces are shortened, we often get more equivalent states, as depicted in $\alpha_1^F(C_3)$, where the traces considered are of length one.

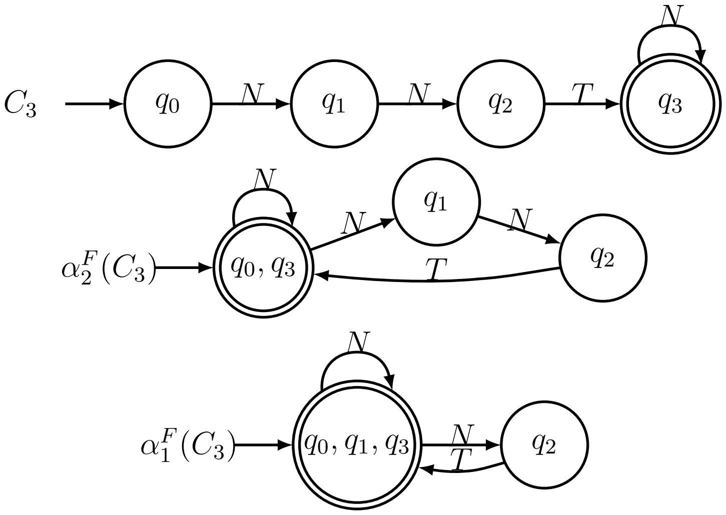
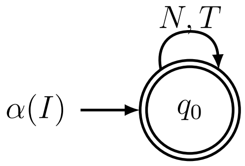


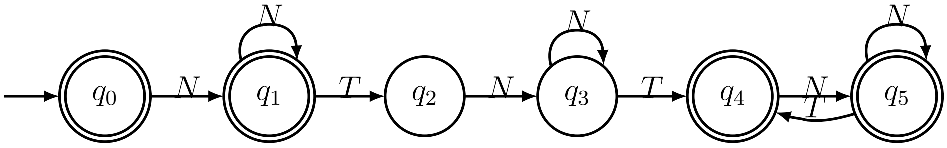
Figure 2.7: Trace abstraction Examples

Next we demonstrate abstraction by predicate languages. Let the predicate language be $L(I \cup B)$ and let us make an abstraction of I by doing predicate language abstraction. We can merge q_0 and q_1 of I, since $L(q_0, I) \cap (I \cup B) \neq \emptyset$ and $L(q_1, I) \cap (I \cup B) \neq \emptyset$ — more precise $L(q_0, I) \cap I \neq \emptyset$ and $L(q_1, I) \cap B \neq \emptyset$ and so our abstraction will result in:

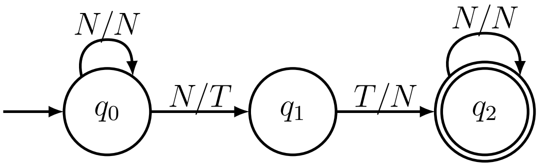


Notice that this abstraction is in fact already violating our requirements defined by B.

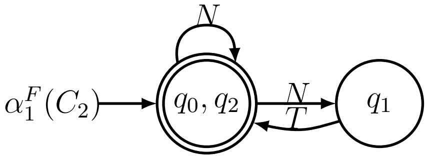
Now let us have a look at how refining abstractions is done. As an example take $\alpha_1^F(C_3)$ from Figure 2.7. This automaton is intersecting with the bad set. The intersecting set is represented by automaton X_3:



So we backtrack, by applying T^{-1},



to X_3 and getting X_2, where in fact $X_2 = \emptyset$ and $\emptyset$ does not intersect with $\alpha_1^F(C_2)$:



And so we have found a spurious counter example and need to refine. The simple way is to simply exclude X_3 from the abstraction function. An other way is to exclude all the intersections cought while backtracking (in our example it is indifferent). △

The article does not give one method universal method for how using the acceleration technique, or the tool. But the article suggests that the initial set and the bad set are good candidates for the predicate language.

Chapter 3

Expresivness of Regular Model Checking

In this section we will present a number of results concerning the expressiveness of regular model checking. First we show that in general *RMC* is undecidable. Then we show that by constraining the initial set to be finite (we call this *RMC-finite*), this problem becomes PSPACE-complete. The last result shows that the strength of regular model checking does not increase, at least for safety questions, by using non-length preserving transducers instead of length preserving ones.

3.1 RMC-verification is undecidable

In the literature several articles claim that regular model checking frameworks are Turing powerful hence *RMC* (that is regular model checking with a length preserving transducer) is undecidable. However to the best of our knowledge, we are not aware of any written proof of this folklore result. For completeness we proof this.

To proof that *RMC* is undecidable we make a reduction from the halting problem of a Two-Counter Minsky Machine to *RMC*.

A Two-Counter Minsky Machine is a simple computation device with three instruction types and two counters. Yet the Minsky Machine is simple it is still Turing powerful [18].

We will use the following definition of a Two-Counter Minsky Machine.

Definition 3.1. A Two-Counter Minsky Machine is a computation device, with a program that can modify the values of two non-negative integer counters, c_1 and c_2.

A Two-Counter Minsky Machine program is a sequence of instructions

$$\begin{array}{rl} 1: & Ins_1 \\ 2: & Ins_2 \\ \vdots & \vdots \\ \ell_{end}-1: & Ins_{\ell_{end}-1} \\ \ell_{end}: & \texttt{HALT} \end{array}$$

where for every i, $1 \leq i < \ell_{end}$. Ins_i is one of the two types:

- c_k`++; goto` j; where $k \in \{1,2\}$, which we denote INC_k, or
- `if` c_k`==0 then goto` j`; else {`c_k`--; goto` m`;}` where $k \in \{1,2\}$, which we denote TAD_k .

The last instruction is always the `HALT` instruction.

A configuration of the Two-Counter Minsky Machine is written as (ℓ, v_1, v_2) where ℓ is the number of the current line in the Two-Counter Minsky Machine program, and v_1 and v_2 are the values of the counters c_1 and c_2 respectively.

The notation $(i, v_1, v_2) \rightarrow (h, v_1', v_2')$ means that we perform the instruction on line i, Ins_i, resulting in configuration (h, v_1', v_2') defined as expected.

If u and u' are configurations of the Two-Counter Minsky Machine then $u \xrightarrow{n} u'$ means that we can reach u' from u, in n steps. Similarly $u \xrightarrow{*} u'$ means that it is possible to reach u' from u with a finite number of steps (including 0). ♢

The halting problem of a Two-Counter Minsky Machine can be formulated as the decision problem $HALT_{minsky}$.

Definition 3.2. ***HALT***$_{minsky}$

Instance: A Two-Counter Minsky Machine M.

Question: Does M reach the halt instruction from the initial configuration $(1, 0, 0)$?

♢

Theorem 3.3 ([18]). *$HALT_{minsky}$ is undecidable.*

3.1.1 Reduction

We will construct a reduction from $HALT_{minsky}$ to RMC.

Given Two-Counter Minsky Machine M we construct a RMC-framework, $R = (I, T, B)$, such that $I \circ T^* \cap B \neq \emptyset$ if and only if the computation of M halts.

We describe the states of M using strings on the form $\ell\#w_1\#w_2\#$, where w_1, w_2 are sub-strings describing the values of the counters c_1, c_2 respectively and '#' is used as a separator symbol between the values. For encoding the values of the counter we use a unary encoding.

The transducer has to be length preserving, but the values of the counter might grow to be arbitrary large. We solves this problem by modifying the initial set. As the initial set is allowed to be infinite, we make sure that there always is a string large enough to write the value of any number.

In general we describe a state of a Two-Counter Minsky Machine (ℓ, v_1, v_2), as the set of strings on the form $\ell\#X^{v_1}\_^*\#X^{v_2}\_^*\#$. This means that there always is at least one string large enough to contain any number.

The initial set I will be $I = L(1\#\_^*\#\_^*)$, which is the encoding of the initial configuration of the Two-Counter Minsky Machine. And the bad set B, will be any string representing the halt instruction: $B = L(\ell_{end}\#X^*\_^*\#X^*\_^*\#)$ where line ℓ_{end} of the program contains the halt instruction.

For each line ℓ in the program we create a transducer T_ℓ based on the instruction type of line ℓ.

The transducer for the instruction type *INC* on c_1 is illustrated in Figure 3.1. The transducer first reads the line number of the current instruction and switches it with the line number of the instruction we jump to. Then it reads the separator symbol #, copies it and the following X symbols until the next blank symbol is read. Then it exchanges the read blank symbol with an X, copies blanks until the next separator symbol. Now we copy the symbols from here until we reach a new # symbol and reach the final state.

Transducer *INC* on c_2 is illustrated in Figure 3.2, which works just like *INC* on c_1, only that we copy the value of the first counter and increase the second counter.

Transducer *TAD* on c_1 is illustrated in Figure 3.3. The transducer works by first choosing non-deterministically whether the counter c_1 is zero or not. The upper branch of the transducer checks if c_1 is zero by copying blank symbols until we read a # symbol and copies from here. The lower part of the figure decreases the counter by one, by converting the last X into a blank symbol. *TAD* on c_2 is illustrated in Figure 3.4 and works just like TAD on c_1, only that counter c_2 is checked and decreased instead.

We let T be the union of the transducers for each instruction in M.

Now the question whether $I \circ T^* \cap B \stackrel{?}{=} \emptyset$ is the same as asking if M reaches the halt instruction.

As we know that $HALT_{minsky}$ is undecidable we know that the problem *RMC* is undecidable too.

This leads to the theorem,

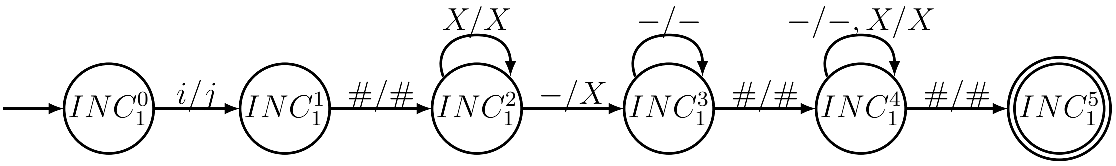


Figure 3.1: The transducer for instructions of the type: c_1`++; goto` j`;`.

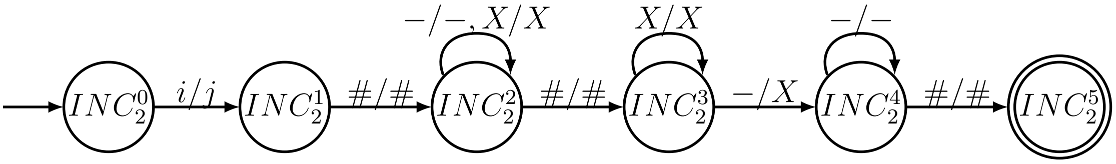


Figure 3.2: The transducer for instructions of the type: c_2`++; goto` j`;`.

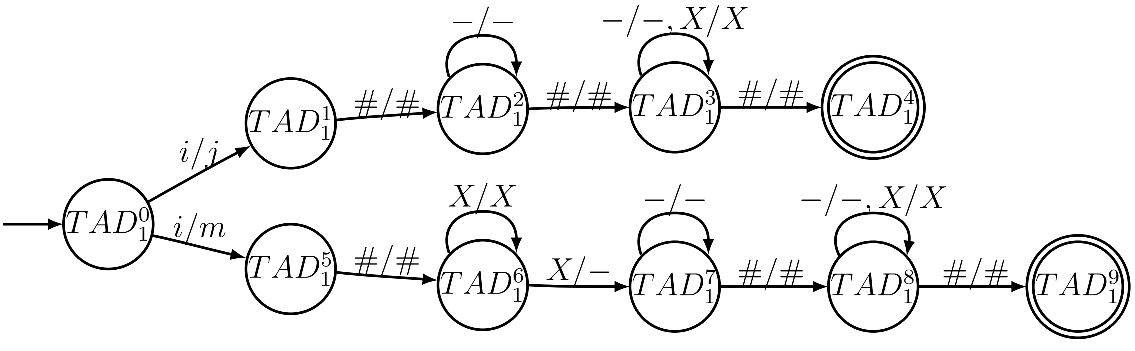


Figure 3.3: The transducer for instructions of the type: `if` c_1`==0 then goto` j`; else{`c_1`--; goto` m`;}` .

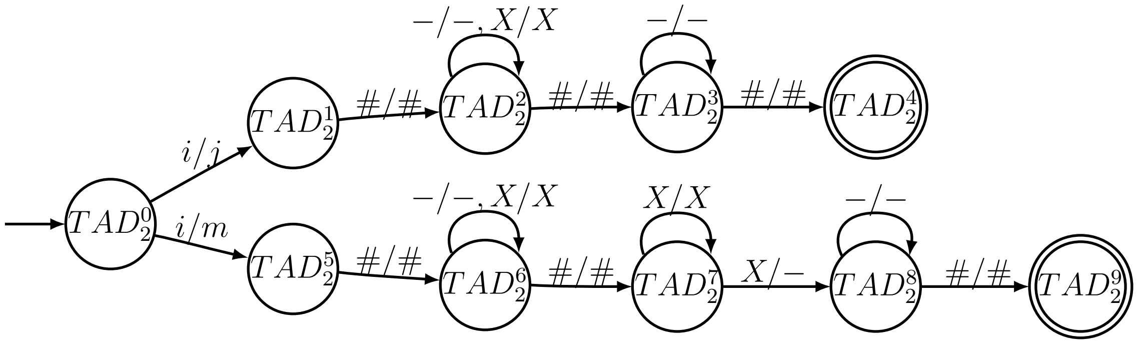


Figure 3.4: The transducer for instructions of the type: `if` c_2`==0 then goto` j`; else{`c_2`--; goto` m`;}` .

Theorem 3.4. *RMC is undecidable.*

3.1.2 Correctness of Reduction

To prove that our reduction is correct we formulate and prove the two stronger lemmas, Lemma 3.5 and Lemma 3.6.

Lemma 3.5. *Given a Two-Counter Minsky Machine M, initial set $I = L(1\#\_^*\#\_^*\#)$, and the length-preserving transducer T constructed from M according to the method presented in section 3.1.1, we have that*

$$\forall n.\forall b.\exists k_1, k_2 \geq b.[(1,0,0) \xrightarrow{n} (y, v_1, v_2) \Rightarrow y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^n].$$

Proof. By induction on n.

Basis step $(n = 0)$
Assume a given b, let $k_1 = k_2 = b$. We know that $(1,0,0) \xrightarrow{0} (1,0,0)$. So it is apparent that

$$(1,0,0) \xrightarrow{0} (1,0,0) \Rightarrow 1\#\_^{k_1}\#\_^{k_2}\# \in I \circ T^0$$

because $I \circ T^0 = I = L(1\#\_^*\#\_^*\#)$ we get that $1\#\_^{k_1}\#\_^{k_2}\# \in I \circ T^0$.

Induction Hypothesis (n)

$$IH(n) \equiv$$

$$\forall b.\exists k_1, k_2 \geq b.[(1,0,0) \xrightarrow{n} (y, v_1, v_2) \Rightarrow y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^n].$$

Induction Step $(n+1)$
Given b we will find $k_1, k_2 \geq b$ s.t.

$$(1,0,0) \xrightarrow{n+1} (y, v_1, v_2) \Rightarrow y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}.$$

If $(1,0,0) \xrightarrow{n+1} (y, v_1, v_2)$ then $(1,0,0) \xrightarrow{n} (y', v'_1, v'_2) \rightarrow (y, v_1, v_2)$. By induction hypothesis we know that for any b' there are

$$k'_1, k'_2 \geq b' \text{ such that } y'X^{v'_1}\_^{k'_1}\#X^{v'_2}\_^{k'_2}\# \in I \circ T^n. \tag{3.1}$$

The label y' can hold an instruction of one of the instruction types *INC*$_1$, *INC*$_2$, *TAD*$_1$ or *TAD*$_2$. We show for each case that the claim holds.

- $INS_{y'} \equiv c_1\texttt{++; goto } y\texttt{;}$ (*INC*$_1$)

  So we have $(y', v'_1, v'_2) \rightarrow (y, v_1, v_2)$ where $v_1 = v'_1 + 1$ and $v_2 = v'_2$. We want to show that for any given b, there are $k_1, k_2 \geq b$ s.t.

  $$y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}.$$

  By using (3.1) where $b' = b + 1$ there are $k'_1, k'_2 \geq b' > b$ such that $s' = y'\#X^{v'_1}\_^{k'_1}\#X^{v'_2}\_^{k'_2}\# \in I \circ T^n$.

  Now by running transducer T on s', the transducer part 3.1 on page 32 applies and so $s'[T]s$, where $s = y\#X^{v'_1+1}\_^{k'_1-1}\#X^{v'_2}\_^{k'_2}\# = y\#X^{v_1}\_^{k'_1-1}\#X^{v_2}\_^{k'_2}\#$. Hence $s \in I \circ T^{n+1}$.

  By choose $k_1 = k'_1 - 1$ and $k_2 = k'_2$ we get $k_1, k_2 \geq b$ and so this proves the claim for this case.

- $INS_{y'} \equiv c_2\texttt{++; goto } y\texttt{;}$ (*INC*$_2$)

  This case is symmetric to *INC*$_1$.

- $INS_{y'} \equiv \texttt{if } c_1\texttt{==}0\texttt{ then goto } j\texttt{; else\{}c_1\texttt{--; goto } m\texttt{;\}}$ (*TAD*$_1$)

  In this case we have $(y', v'_1, v'_2) \rightarrow (y, v_1, v_2)$, where
  $$(y, v_1, v_2) = \begin{cases} (j, v'_1, v'_2) & \text{if } v'_1 = 0 \\ (m, v'_1 - 1, v'_2) & \text{if } v'_1 > 0 \end{cases}.$$
  By using (3.1) where $b' = b$ there are $k'_1, k'_2 \geq b' \geq b$ such that $s' = y'\#X^{v'_1}\_^{k'_1}\#X^{v'_2}\_^{k'_2}\# \in I \circ T^n$.

Now by running transducer T on s', the transducer part 3.3 on page 32 applies and so $s'[T]s$, where $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\#$, where $k_2 = k_2'$, $k_1 = \begin{cases} k_1' & \text{if } v_1' = 0 \\ k_1' + 1 & \text{if } v_1' > 0 \end{cases}$, $v_2 = v_2'$ and $v_1 = \begin{cases} v_1' & \text{if } v_1' = 0 \\ v_1' - 1 & \text{if } v_1' > 0 \end{cases}$.

This implies that $k_1, k_2 \geq b$, and because $s \in I \circ T^{n+1}$ we have proved our claim for this case.

- *INS*$_{y'}$ $\equiv$ `if` c_2`==0 then goto` j`; else{`c_2`--; goto` m`;}` (*TAD$_2$*)

  This part is symmetric to *TAD$_1$*.

□

Observe that based on the transducer and initial set given by the reduction, all strings in $I \circ T^n$ for any n will be of the form $y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}$ for any $v_1, k_1, v_2, k_2 \in \mathbb{N}_0$.

Lemma 3.6. *Given a Two-Counter Minsky Machine M, initial set $I = L(1\#\_^*\#\_^*\#)$ and the length-preserving transducer T constructed from M according to the method presented in section 3.1.1, we have that*

$$\forall n. \forall k_1, k_2. [y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^n \Rightarrow (1,0,0) \xrightarrow{n} (y, v_1, v_2)].$$

Proof. By induction on n.

Basis step $(n = 0)$
We show that for all $k_1, k_2 \in \mathbb{N}$, if $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^0$ then $(1,0,0) \xrightarrow{0} (y, v_1, v_2)$.

As $I \circ T^0 = I = L(1\#\_^*\#\_^*\#)$ we know that $v_1 = v_2 = 0$. Now we need to show that $(1,0,0) \xrightarrow{0} (1,0,0)$.

This is trivially true and so the basis step is shown.

Induction Hypothesis (n)

$$IH(n) \equiv \forall k_1, k_2. [y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^n \Rightarrow (1,0,0) \xrightarrow{n} (y, v_1, v_2)].$$

Induction step $(n+1)$

We show that for all $k_1, k_2 \in \mathbb{N}_0$ we have that

$$y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1} \Rightarrow (1,0,0) \xrightarrow{n+1} (y, v_1, v_2).$$

Because $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}$ there must be a string $s' = y'\#X^{v_1'}\_^{k_1'}\#X^{v_2'}\_^{k_2'}\# \in I \circ T^{n}$ s.t. $s'[T]s$.

By using the induction hypothesis we have that

$$(1,0,0) \xrightarrow{n} (y', v_1', v_2').$$

If $s' = y'\#X^{v_1'}\_^{k_1'}\#X^{v_2'}\_^{k_2'}\# \in I \circ T^{n}$ translates to $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}$ by T, then different parts of T will apply to s'. They will be of the following types: INC_1, INC_2, TAD_1 or TAD_2 as illustrated in Figure 3.1, 3.2, 3.3 and 3.4 respectively. We show that for each case the claim holds.

- INC_1 (c_1`++; goto` j`;`):

  Let $s' = y'\#X^{v_1'}\_^{k_1'}\#X^{v_2'}\_^{k_2'}\# \in I \circ T^{n}$ and $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}$ s.t. $s'[T]s$ where $v_1 = v_1' + 1$ and $v_2 = v_2'$ and $k_1 = k_1' - 1$ and $k_2 = k_2'$. We have to show that

  $$(y', v_1', v_2') \rightarrow (y, v_1' + 1, v_2) = (y, v_1, v_2).$$

  We see that the Two-Counter Minsky Machine instruction: $\mathit{INS}_{y'} \equiv$ c_1`++; goto` y`;` corresponds to $(y', v_1', v_2') \rightarrow (y, v_1, v_2)$, which proves the claim for this case.

- INC_2 (c_2`++; goto` j):

  This case is symmetric to INC_1.

- TAD_1 (`if` c_1`==`0 `then goto` j`; else{`c_1`--; goto` m`;}`):

  In this case let $s' = y'\#X^{v_1'}\_^{k_1'}\#X^{v_2'}\_^{k_2'}\# \in I \circ T^{n}$ and $s = y\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2}\# \in I \circ T^{n+1}$ s.t. $s'[T]s$, where $v_1 = \begin{cases} v_1' & \text{if } v_1' = 0 \\ v_1' - 1 & \text{if } v_1' > 0 \end{cases}$, $v_2 = v_2'$, $k_1 = \begin{cases} k_1' & \text{if } v_1' = 0 \\ k_1' + 1 & \text{if } v_1' > 0 \end{cases}$ and $k_2 = k_2'$.

  Now by executing $\mathit{INS}_{y'} \equiv$ `if` c_1`==`0 `then goto` j`; else{`c_1`--; goto` m`;}`. We see that $(y', v_1', v_2) \rightarrow (y, v_1, v_2)$, where $y = \begin{cases} j & \text{if } v_1' = 0 \\ m & \text{if } v_1' > 0 \end{cases}$, which proves the claim for this case.

- *TAD₂* (`if` c_2`==0 then goto` j`; else{`c_2`--; goto` m`;}`):

  This case is symmetric to *TAD₁*.

□

Given a Minsky Machine M and *RMC*-framework $R = (I, T, B)$ created according to our reduction. By Lemma 3.6 we know that, if the string $\ell\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2} \in I \circ T^*$ for some $v_1, v_2, k_1, k_2 \in \mathbb{N}$, then the computation of M can reach configuration (ℓ, v_1, v_2) from $(1, 0, 0)$.

By the construction of T we know that $I \circ T^*$ only contains strings on the form $i\#X^{v_1}\_^{k_2}\#X^{v_2}\_^{k_2}$ for $v_1, v_2, k_1, k_2 \in \mathbb{N}$.

By Lemma 3.5 we know that if M can reach configuration (ℓ, v_1, v_2) from $(1, 0, 0)$, then $\ell\#X^{v_1}\_^{k_1}\#X^{v_2}\_^{k_2} \in I \circ T^*$ for some $k_1, k_2 \in \mathbb{N}$.

By combining this we get that

$$I \circ T^* \cap B \neq \emptyset \iff (1, 0, 0) \xrightarrow{*} (\ell_{end}, v_1, v_2).$$

This was what we wanted to show, and Theorem 3.4 is hereby proved.

3.2 RMC-finite is PSPACE-complete

In section 3.1 we showed that in general *RMC* is undecidable. However, by constraining the initial set so that it is finite, we get that the problem (*RMC-finite*) is decidable and PSPACE-complete. Recall *RMC-finite* defined in Section 2.1.1, where the initial set is finite. To show *RMC-finite* is PSPACE-complete we show that

- *RMC-finite* is in PSPACE, and
- there is a polynomial time reduction from the reachability problem in one-safe unary Petri Net to *RMC-finite*.

In the reduction we use one-safe unary Petri Nets, so we will start by having a look at these.

3.2.1 Petri Net

Definition 3.7. A *Petri Net* is a 4-tuple: $N = (P, T, F, M_0)$ where

- P is a finite set of places,

- T is a finite set of transitions,
- F is a flow relation: $F \subseteq (P \times T) \cup (T \times P)$, and
- $M_0 : P \to \mathbb{N}_0$ is the initial marking of N.

Let $N = (P, T, F, M_0)$ be a Petri Net. A marking of N is a function $M : P \to \mathbb{N}_0$. The function $M(p)$ returns the number of tokens in place p. Assuming $P = \{p_1, p_2, \cdots, p_m\}$, then we can write a marking of Petri Net N as the vector $(M(p_1), M(p_2), \ldots, M(p_m))$.

Given $a \in P \cup T$. The preset of a is ${}^\bullet a = \{a' | a'Fa\}$ and the postset of a is $a^\bullet = \{a' | aFa'\}$.

In Petri Net $N = (P, T, F, M_0)$ a transition $t \in T$ is enabled at marking M if all places $p \in {}^\bullet t$ satisfy $M(p) > 0$.

Given a transition $t \in T$ we define a relation $\xrightarrow{t}$ between markings as: $M \xrightarrow{t} M'$ if t is enabled at M and for every place p, $M'(p) = M(p) + F(t, p) - F(p, t)$, where $F(x, y)$ is 1 if $(x, y) \in F$ else it is 0. We use the notation $M \xrightarrow{k} M'$ to denote that marking M' can be reached from marking M in k number steps. We write $M \xrightarrow{*} M'$ if M' is reachable from M in any finite number of steps.

A marking in N is *one-safe* if, for every place p in N it holds that $M(p) \leq 1$. A Petri Net N is called *one-safe* if all reachable marking in N are one-safe. A Petri Net is called *unary* if at most one transition is enabled for each reachable marking. ◊

Example 3.8. A Petri Net can be drawn as in Figure 3.5. A circle defines a place in the net. A square defines a transition. The arrows define the presets and postsets of places and transitions. A dot in a place means that this place holds a token. The net in Figure 3.5 is one safe, as all reachable marking holds at most one token in each place. The net is unary as all reachable markings only enable at most one transition.

△

The problem of reachability in a one-safe unary Petri net can be formulated as the decision problem *1S-PN-reach*.

Definition 3.9. *1S-PN-reach*

Instance: A one-safe unary Petri Net $N = (P, T, F, M_0)$ and marking of M_r.

Question: Is M_r reachable from marking M_0?

◊

Theorem 3.10 ([9])**.** *1S-PN-reach is PSPACE-complete*

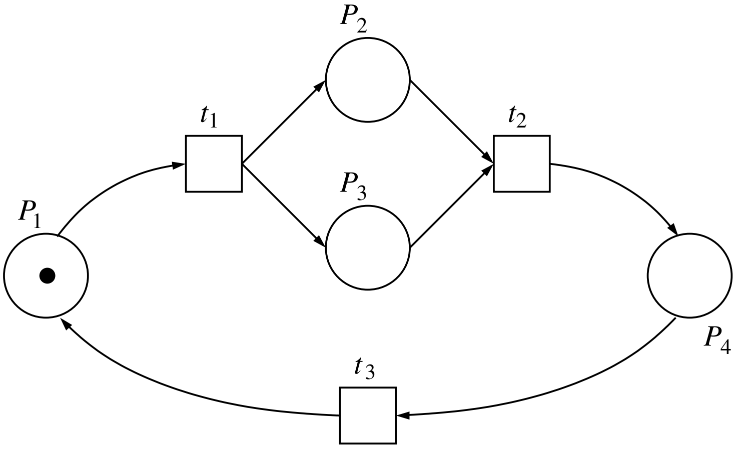


Figure 3.5: Example of a Petri Net

3.2.2 PSPACE Algorithm

First we show that *RMC-finite* belongs to PSPACE by providing Algorithm 1, that decides *RMC-finite* in PSPACE.

Algorithm 1: Algorithm to decide *RMC-finite*.

```
Input: RMC-finite-framework $R = (I, T, B)$.
Output: True if $I \circ T^* \cap B \neq \emptyset$, false otherwise.
begin
    Nondeterministically select a string $s \in I$ ;
    while $s \notin B$ do
        Let $s := s'$, where $s'$ is a nondeterministically chosen string s.t.
        $(s, s') \in [T]$. ;
        if s' does not exist then
            return false
    return true
end
```

This algorithm does not terminate, however by using a trick from Savitch's theorem,[24] we can detect if the machine loops, and then terminate. A loop can be detected by using no more than NPSPACE.

The algorithm uses at most $O(n)$ space in its computation, where n is the size of the input. The algorithm uses space to store the input of the algorithm. Beyond that two strings s and s' are stored for each non-deterministic choice of string. The total size of the two strings is $|s| + |s'|$, where the size of s' does not exceed the size of s for any s, as the transducer is length-preserving (this of coarse also holds for any string derived from a string from the initial set I). The algorithm therefore belongs to NPSPACE, and as NPSPACE = PSPACE [24] we have an algorithm deciding *RMC-finite* in PSPACE.

3.2.3 Reduction

We now give a reduction from *1S-PN-reach* to *RMC-finite*.

The idea is to create RMC-framework $R = (I, T, B)$, for a given one-safe unary Petri Net N and goal marking M_r, such that $I \circ T^* \cap B \neq \emptyset$ iff $M_0 \xrightarrow{*} M_r$.

Given a Petri Net $N = (P_N, T_N, F_N, M_0)$ where $P_N = \{p_1, p_2, \ldots, p_m\}$ and a marking M_r, we describe the markings of the Petri Net by a string $s = a_1a_2 \cdots a_m$. Here a_i corresponds to the number of tokens in place $p_i \in P_N$, for $1 \leq i \leq m$. As N is 1-safe, we only need two symbols, $\Sigma = \{0, 1\}$ to describe the possible markings. We write $\langle M \rangle$ to denote the string representation of the marking $M = (a_1, a_2, \ldots, a_m)$ as the string $a_1a_2 \cdots a_m$.

For an *RMC-finite*-framework $R = (I, T, B)$, we let $I = \{\langle M_0 \rangle\}$ and $B = \{\langle M_r \rangle\}$. The flow relation of the Petri Net can be expressed using a length preserving finite-state transducer T. Where for each $t \in T_N$ we create a transducer modeling the transition in the Petri Net.

The algorithm for the reduction is shown in Algorithm 2.

The running time of Algorithm 2 is as follows: for each transition in T_N we create at most one transducer of linear size in the number of places in N. This gives a running time $O(|T_N| \cdot |P_N|)$. This belongs to PTIME.

We have hereby shown a polynomial time reduction from *1S-PN-reach* to *RMC-finite*. To illustrate the reduction we give the following example.

Example 3.11. Given a one-safe unary Petri Net N defined as in Figure 3.5 and resulting marking $M_r = (0, 0, 0, 1)$. We can create a *RMC-finite*-instance $R = (I, T, B)$ using Algorithm 2. The initial set I will be $I = \{1000\}$, as the Petri Net has 4 places and a token in place 1. The set of "bad states" will be $B = \{0001\}$. To construct T three transducers will be created.

The transducer created from transition t_1 is illustrated in Figure 3.6. First it reads 1 and replaces it with 0 as one token is consumed in place p_1. It then reads 0 and replaces it with 1, as one token is created in p_2, and reads 0 and replaces it with 1 as a token is created in p_3. Rest of the string is left unchanged.

The transducer created from transition t_2 is illustrated in Figure 3.6. It reads 0 and outputs 0 or reads 1 and outputs 1, as the number of tokens

Algorithm 2: Algorithm for construction the reduction from *1S-PN-reach* to *RMC-finite*

```
Input: A one-safe unary Petri Net
       $N = (P_N = \{p_1, p_2, \ldots, p_m\}, T_N, F_N, M_0)$. A marking $M_r$
Output: A RMC-finite-framework $R = (I, T, B)$
begin
    $I := \{\langle M_0 \rangle\}$;
    $B := \{\langle M_r \rangle\}$;
    for each $t \in T_N$ do
        construct transducer $T_t = (Q^t, \Sigma, \delta^t, q_0^t, F^t)$, where
        $Q^t := \{q_0^t, q_1^t ... q_m^t\}$ ;
        $\Sigma := \{1, 0\}$ ;
        $F := \{q_m^t\}$;
        for $i = \{0, 1, \ldots, m-1\}$ do
            Add rule $q_i^t, \xrightarrow{X_i} q_{i+1}^t$ to $\delta^t$ where
            $X_i = \begin{cases} 1/0 & \text{if } p_i \in {}^\bullet t \backslash t^\bullet \\ 0/1 & \text{if } p_i \in t^\bullet \backslash {}^\bullet t \\ 1/1 & \text{if } p_i \in {}^\bullet t \cap t^\bullet \\ 1/1, 0/0 & \text{otherwise} \end{cases}$
    $T := \bigcup_{t \in T_N} T_t$;
end
```

in place p_1 is left unchanged. It then reads 1 and outputs 0 as one token is consumed from place p_2 in transition t_2, again it reads 1 and outputs 0 as one token is consumed from place p_3. At last it reads 0 and output 1, as one token is created in place p_4.

The transducer created from transition t_3 is illustrated in Figure 3.6.

△

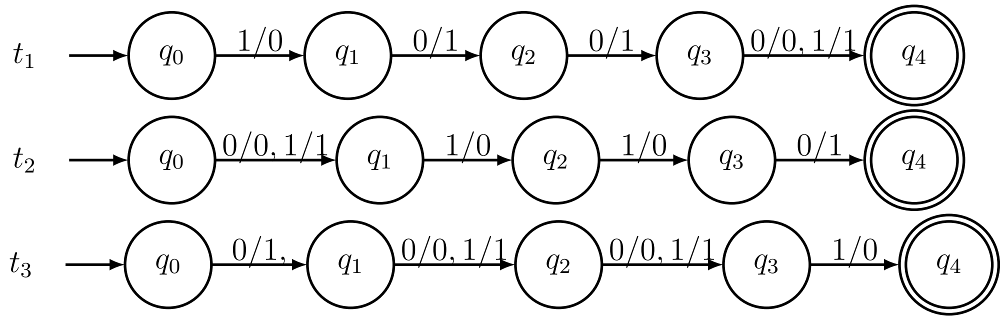


Figure 3.6: The transducer created from transitions t_1, t_2 and t_3 by Algorithm 2 on the Petri Net in Figure 3.5

3.2.4 Correctness of Reduction

To show the correctness of the reduction we prove Theorem 3.12

Theorem 3.12. *Given one-safe unary Petri Net $N = (P_N, T_N, F_N, M_0)$, marking M_r and regular model checking-framework $R = (I, T, B)$ constructed according to Algorithm 2 on page 41, we have*

$$I \circ T^n \cap B \neq \emptyset \iff M_0 \xrightarrow{n} M_r.$$

Lemma 3.13. *All strings in $I \circ T^n$ are on the form $a_1 a_2 \cdots a_m$.*

Lemma 3.13 is true as the transducer always produces strings on this form.

To show Theorem 3.12 we formulate and prove the stronger lemma,

Lemma 3.14. *Given one-safe unary Petri Net $N = (P_N = \{p_1, p_2, \ldots, p_i\}$, $T_N, F_N, M_0 = (a_1, \ldots, a_m))$, marking $M_r = (b_1, \ldots, b_m)$ and regular model checking-framework $R = (I, T, B)$ constructed according to Algorithm 2 on page 41, then*

$$(a_1, \ldots, a_m) \xrightarrow{n} (b_1, \ldots, b_m) \iff b_1 \cdots b_m \in I \circ T^n.$$

Proof. By induction on n.

We show the two directions separately. We start by showing the right direction.

"$\Longrightarrow$ direction"

Basis step $(n = 0)$

We have to show that if $(a_1, \ldots, a_m) \xrightarrow{0} (b_1, \ldots, b_m) = (a_1, \ldots, a_m)$ then $a_1 \cdots a_m \in I \circ T^0$.

Because $I \circ T^0 = I = \{\langle M_0 \rangle\}$ we get that $a_1 \cdots a_m \in I \circ T^0$ and this shows the basis step.

Induction Hypothesis (n)

$$IH(n) \equiv (a_1, \ldots, a_m) \xrightarrow{n} (b_1, \ldots, b_m) \implies b_1 \cdots b_m \in I \circ T^n$$

Induction Step $(n+1)$
We have to show that if $(a_1,\dots,a_m) \xrightarrow{n+1} (b_1,\dots,b_m)$ then

$$b_1 \cdots b_m \in I \circ T^{n+1}$$

If $(a_1,\dots,a_m) \xrightarrow{n+1} (b_1,\dots,b_m)$ then $(a_1,\dots,a_m) \xrightarrow{n} (b'_1,\dots,b'_m) \xrightarrow{t} (b_1,\dots,b_m)$ then by $IH(n)$ we have that $s' = b'_m \cdots b'_m \in I \circ T^n$

Now by running transducer T on s', the transducer part created from transition t applies, and $s'[T]s$ where $s = a_1 \cdots a_m \in I \circ T^{n+1}$.

"$\Longleftarrow$ direction"

Basis step $(n=0)$
We need to show that if $b_1 \cdots b_m \in I \circ T^0$ then $(a_1,\dots,a_m) \xrightarrow{0} (b_1,\dots,b_m)$.

As $I \circ T^n = I = \{a_1 \cdots a_m\}$ and as $(a_1,\dots,a_m) \xrightarrow{0} (a_1,\dots,a_m)$ we have shown the base case.

Induction Hypothesis (n)
Recall Lemma 3.13, then:

$$IH(n) \equiv b_1 \cdots b_m \in I \circ T^n \implies (a_1,\dots,a_m) \xrightarrow{n} (b_1,\dots,b_m).$$

Induction Step $(n+1)$
We have to show that, if $s = b_1 \cdots b_m \in I \circ T^{n+1}$ then $(a_1,\dots,a_m) \xrightarrow{n+1} (b_1,\dots,b_m)$.

If $b_1 \cdots b_m \in I \circ T^{n+1}$ then we know that there is a string $s' = b'_1 \cdots b'_m \in I \circ T^n$, such that $s'[T]s$. By $IH(n)$ we know that $(a_1,\dots,a_m) \xrightarrow{n} (b'_1,\dots,b'_m)$.

Now we need to show that $(b'_1,\dots,b'_m)$ in one step can reach $(b_1,\dots,b_m)$. When $s'[T]s$ we now that, one part of the transducer corresponding to one transition in the Petri Net. As the Petri Net is unary, only one transition is enabled. By choosing this transition we get $(b'_1,\dots,b'_m) \xrightarrow{t} (b_1,\dots,b_m)$. This shows the $\Leftarrow$ direction.

□

As Lemma 3.14 is stronger than Theorem 3.12 we have proven the theorem.

3.3 Expressiveness on Non-Length Preserving Transducers

Most literature in the area concern almost exclusively length preserving transducers. The only article to our knowledge that deals explicitly with non-length preserving transducers is [10].

By modelling a non-length preserving regular model checking-framework using a length-preserving regular model checking-framework we can show that surprisingly we do not add extra expressive power, by using non-length preserving transducers, at least for safety properties.

Recall the decision problems *RMC* and *GRMC* defined in Section 2.1.1. We show how to reduce from *GRMC* to *RMC*. The other way is trivially true, as of coarse a length preserving transducer is a non-length preserving transducer.

3.3.1 Reduction

Given a general (non length-preserving) RMC-framework $R_g = (I_g, T_g, B_g)$ we create a (length-preserving) *RMC*-framework $R_\ell = (I_\ell, T_\ell, B_\ell)$ s.t.

$$I_g \circ T_g^* \cap B_g \neq \emptyset \iff I_\ell \circ T_\ell^* \cap B_\ell \neq \emptyset.$$

A non-length preserving transducer is allowed to grow or shrink a string, using ε-transitions. The idea is to describe this with a length preserving transducer. This is done by modelling the empty string 'ε' explicitly by using a fresh blank symbol '_'. By exchanging all ε-transition in the non-length preserving transducer with transitions with the blank symbol, we have created a length preserving transducer. This will work in the cases where the computation if finite, which it will be when searching for safety problem. In the case of infinite computations we have no grantee for this approach to work. We will discuss these details in later sections.

To model that between every symbol we can insert an arbitrary number of symbols, we change the initial set to contain an arbitrary number of blank symbols between any of its symbols. As the initial set is allowed to be infinite but regular, we can describe it by an NFA for I_g where all states have been added a blank symbol ('_') loop. Now we know, that for each string $a_1 a_2 \cdots a_m \in I_g$ we have a set of strings in I described by the regular expression $\_^* a_1 \_^* a_2 \_^* \cdots \_^* a_m \_^*$ in I_g. For any string in I_ℓ we always have an other string with a larger number of blanks.

The function φ (Algorithm 3 on the next page) translates the strings of a regular set to a set of strings containing blank symbols between all its symbols, as described above.

Algorithm 3: Function φ

```
Input: A regular set described by a non-deterministic finite
       automaton $M = (Q, \Sigma, q_0, F, \delta)$.
Output: A regular set described by a non-deterministic finite
        automaton $\varphi(M) = (Q, \Sigma \uplus \{\_\}, q_0, F, \delta')$.
begin
    $\delta'(g, a) := \delta(g, a) \quad \forall g \in Q$ and $\forall a \in \Sigma$
    $\delta'(g, \_) := \{g\} \quad \forall g \in Q$
end
```

The function Ψ (Algorithm 4) translates a non-length preserving transducer into a length preserving one. (Recall that by Definition 2.5 of a transducer we have $\varepsilon \notin \Sigma$.)

Algorithm 4: Functions Ψ.

```
Input: A finite-state transducer $T = (Q, \Sigma, q_0, F, \delta)$.
Output: A length preserving finite-state transducer
        $\Psi(T) = (Q, \Sigma \uplus \{\_\}, q_0, F, \delta')$.
begin
    $\delta'(g, a, b) := \delta(g, a, b) \quad \forall g \in Q$ and $\forall a, b \in \Sigma$
    $\delta'(g, a, \_) := \delta(g, a, \varepsilon) \quad \forall g \in Q$ and $\forall a \in \Sigma$
    $\delta'(g, \_, b) := \delta(g, \varepsilon, b) \quad \forall g \in Q$ and $\forall b \in \Sigma$
    $\delta'(g, \_, \_) := \delta(g, \varepsilon, \varepsilon) \cup \{g\} \quad \forall g \in Q$
end
```

We will now give an example of the reduction.

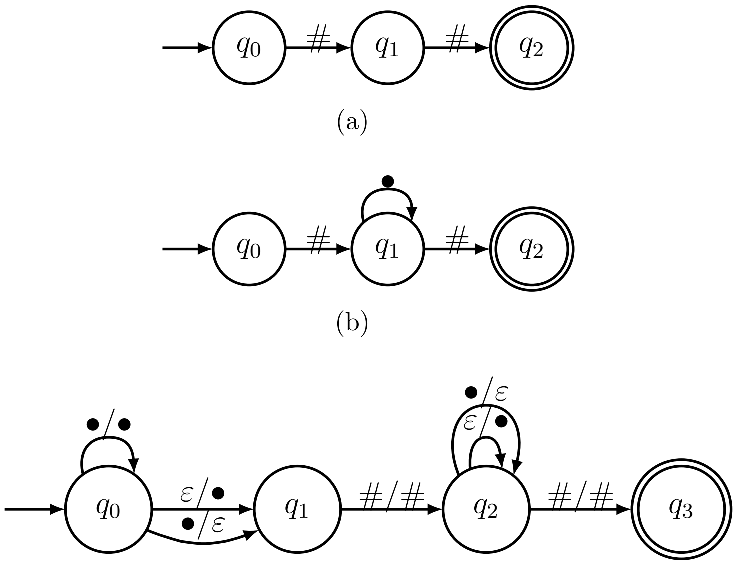
Example 3.15. Given a *GRMC*-framework $R = (I, T = (Q, \Sigma, q_0, F, \delta), B)$ where I is given by Figure 3.7(a), B is given by Figure 3.7(b) and T is given by Figure 3.7(c).

Running φ on the NFA illustrated in Figure 3.7(a) and 3.7(b) results in the NFA in Figure 3.8(a) and Figure 3.8(b), respectively. As every state has a blank symbol loop we know that all strings will be of the form

$$\_^{k_0} a_1 \_^{k_1} a_2 \_^{k_2} \cdots \_^{k_{n-1}} a_m \_^{k_m} \ .$$

Applying Ψ on the transducer illustrated in Figure 3.7(c) results in the transducer illustrated in Figure 3.8(c). All ε have been converted to blank symbols, and each state got added a blank symbol loop.

$\triangle$

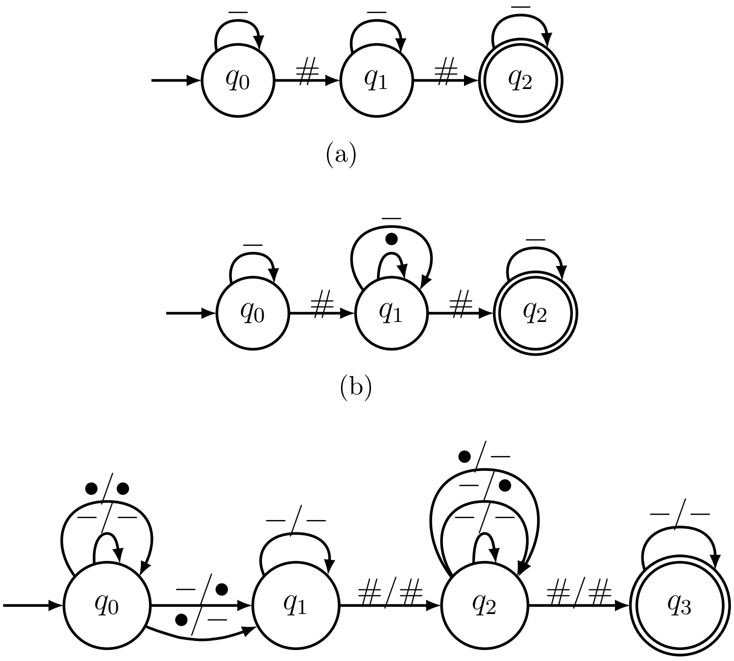


(a)

(b)

(c)

Figure 3.7: General RMC framework, with (a) initial set, (b) bad set and (c) transducer.



(a)

(b)

(c)

Figure 3.8: A RMC framework, with (a) initial set, (b) bad set and (c) transducer.

Theorem 3.16. *Given a GRMC-framework $R_g = (I_g, T_g, B_g)$ and a RMC-framework $R_\ell = (I_\ell, T_\ell, B_\ell)$ created by the method described in section 3.3.1*

$$I_g \circ T_g^* \cap B_g \neq \emptyset \iff I_\ell \circ T_\ell^* \cap B_\ell \neq \emptyset$$

The time complexity of Algorithm 3 and 4 for the reduction is linear in the size of the input. The size of the output has at most grown in linear size of the input, as the only difference are the blank symbol loops.

3.3.2 Correctness of Reduction

To prove Theorem 3.16 we formulate and prove two stronger lemmas.

Lemma 3.17. *Let T be a non-length preserving transducer and I be a set described by a finite automaton. Then*

$$\forall n. \forall b. \forall a_1 \cdots a_m \in I \circ T^n. \exists k_0, \ldots, k_m \geq b.$$

$$\left[\_^{k_0} a_1 \_^{k_1} \cdots \_^{k_{m-1}} a_m \_^{k_m} \in \varphi(I) \circ \Psi(T)^n\right].$$

Proof. By induction on n.

Basis Step $(n = 0)$
Assume a given b.

We have to show that if $a_1 \cdots a_m \in I \circ T^0$ then there are $k_0, \ldots, k_m \geq b$ s.t. $\_^{k_0} a_1 \_^{k_1} \cdots \_^{k_{m-1}} a_m \_^{k_m} \in \varphi(I) \circ \Psi(T)^0$.

As $\varphi(I) \circ \Psi(T)^0 = \varphi(I) \supseteq L(\_^* a_1 \_^* \cdots \_^* a_m \_^*)$ it is apparent that for $k_0 = k_1 = \ldots = k_m = b$ we get $\_^{k_0} a_1 \_^{k_1} \cdots \_^{k_{m-1}} a_m \_^{k_m} \in \varphi(I) \circ \Psi(T)^0$. This shows the base case.

Induction Hypothesis (n)
$IH(n) \equiv$

$$\forall b. \forall a_1 \cdots a_m \in I \circ T^n. \exists k_0, \ldots, k_m \geq b. \left[\_^{k_0} a_1 \_^{k_1} \cdots \_^{k_{m-1}} a_m \_^{k_m} \in \varphi(I) \circ \Psi(T)^n\right]$$

Induction Step $(n + 1)$
Assume a given b, and a given string $s = a_1 \cdots a_m \in I \circ T^{n+1}$. We will find $k_0, \ldots, k_m \geq b$ such that

$$s_\varphi = \_^{k_0} a_1 \_^{k_1} \cdots \_^{k_{m-1}} a_m \_^{k_m} \in \varphi(I) \circ \Psi(T)^{n+1}.$$

We know that if $s \in I \circ T^{n+1}$ then there exists a string $s' = a'_1 \cdots a'_z \in I \circ T^n$, s.t $s'[T]s$.

Now by $IH(n)$ we know that for all b' there exists

$$k'_0, \dots, k'_\ell \geq b' \text{ s.t. } s'_\varphi = \_^{k'_0} a'_1 \_^{k'_1} \cdots \_^{k'_{z-1}} a'_z \_^{k'_z} \in \varphi(I) \circ \Psi(T)^n.$$

We now find $k_0, \dots, k_m \geq b$ such that there is a translation $s'_\varphi[T]s_\varphi$.

As $s'[T]s$ we know that there exists a finite computation of the form $C \equiv q_0 \xrightarrow{\alpha'_1/\alpha_1} q_1 \xrightarrow{\alpha'_2/\alpha_2} \cdots \xrightarrow{\alpha'_t/\alpha_t} q_f$ where $q_0, \dots, q_f \in Q_g$ and $q_f \in F_g$ such that $s' = \alpha'_1 \cdots \alpha'_t$ and $s = \alpha_1 \cdots \alpha_t$ (remember that some α' and α might be equal to ε). We see that t is the number of steps in the computation.

Now by choosing $b' = b + t$, and by using IH on s', we know that there are always enough blank symbols between any two letters to cover the worst case, where the string only grows between these two symbols.

We now show that there is a translation $s'_\varphi[\Psi(T)]s_\varphi$. The rules in the computation C will be of the form $q_w \xrightarrow{a/b} q_x$, $q_w \xrightarrow{a/\varepsilon} q_x$, $q_w \xrightarrow{\varepsilon/b} q_x$ or $q_w \xrightarrow{\varepsilon/\varepsilon} q_x$. For each rule in the computation C we can use a number of steps to match the computation in $\Phi(T)$.

- If $q_w \xrightarrow{a/b} q_x$:
  We know the input symbol can be either a blank or a symbol a. While the input symbol is a blank, we repeatedly apply the rule $q_w \xrightarrow{\_/\_} q_w$. We know that after matching a number of blanks the symbol a will come. We know that $\Psi(T)$ has rule $q_w \xrightarrow{a/b} q_x$, so we can match with the same rule.

- $q_w \xrightarrow{a/\varepsilon} q_x$:
  We know that there can be blank symbols before the symbol a so we match them with the rule $q_w \xrightarrow{\_/\_} q_w$. When we reach the symbol a, we can match it by the rule $q_w \xrightarrow{a/\_} q_x$.

- $q_w \xrightarrow{\varepsilon/b} q_x$:
  We can match this step by the rule $q_w \xrightarrow{\_/b} q_x$.

- $q_w \xrightarrow{\varepsilon/\varepsilon} q_x$:
  We can match it by the rule $q_w \xrightarrow{\_/\_} q_x$

We have now shown the corresponding computation in $\Psi(T)$, and we know that $k_1, \dots, k_n \geq b$ as at most t consecutive spaces where consumed, because t is the length of the computation C.

By this we have proven Lemma 3.17.

□

Lemma 3.18. *Let T be a non-length preserving transducer and I be a set described by a finite automaton. Then*

$$\forall n.\forall k_0,\ldots,k_m.\forall \_^{k_0}a_1\_^{k_1}\cdots\_^{k_{m-1}}a_m\_^{k_m} \in \varphi(I)\circ\Psi(T)^n.\,[a_1\cdots a_m \in I\circ T^n]\,.$$

Proof. By induction on n.

Basis Step $(n = 0)$
We have to show that if $\_^{k_0}a_1\_^{k_1}\cdots\_^{k_m-1}a_m\_^{k_m} \in \varphi(I)\circ\Psi(T)^0$, then $a_1\cdots a_m \in I\circ T^0$.

Because $\_^{k_0}a_1\_^{k_1}\cdots\_^{k_m-1}a_m\_^{k_m} \in \varphi(I)\circ\Psi(T)^0 = \varphi(I)$ we need to show $a_1\cdots a_m \in I\circ T^0$.

As $I\circ T^0 = I$ it is apparent that $a_1\cdots a_m \in I$.

Induction Hypothesis (n)
$IH(n) \equiv$

$$\forall k_0,\ldots,k_m.\_^{k_0}a_1\_^{k_1}\cdots\_^{k_{m-1}}a_m\_^{k_m} \in \varphi(I)\circ\Psi(T)^n.\,[a_1\cdots a_m \in I\circ T^n]$$

Induction Step $(n+1)$

Assume given $k_0,\ldots,k_m$. Then we have to show that if $s = \_^{k_0}a_1\_^{k_1}\cdots\_^{k_{m-1}}a_m\_^{k_m} \in \varphi(I)\circ\Psi(T)^{n+1}$ then $u = a_1\cdots a_m \in I\circ T^{n+1}$.

We know that as s is in $\varphi(I)\circ\Psi(T)^{n+1}$ then there exists a string $s' = \_^{k'_0}a'_1\_^{k'_1}\cdots\_^{k'_{m'-1}}a'_{m'}\_^{k'_{m'}} \in \varphi(I)\circ\Psi(T)^n$, s.t. $s'[\Psi(T)]s$.

Then by $IH(n)$ we have that

$$u' = a'_1\cdots a'_m \in I\circ T^n.$$

Now we have to show that $u'[T]u$.

We know that there exists a computation for $s'[\Psi(T)]s$ of the form: $q_0 \xrightarrow{\alpha'_1/\alpha_1} q_1 \xrightarrow{\alpha'_2/\alpha_2} \cdots \xrightarrow{\alpha'_t/\alpha_t} q_f$, where $q_0,\ldots,q_f \in Q_l$ and $q_f \in F_l$. We now find a computation for $u'[T]u$, by removing all steps on the form $q \xrightarrow{\_/\_} q$ and by exchanging all blank symbols with ε, in the computation of $s'[\Psi(T)]s$. By this we have proved Lemma 3.18.

□

By proving Lemma 3.18 and 3.17 we know that for each step in the computation the two frameworks can simulate each other; and so we have proved Theorem 3.16.

3.3.3 About the Result

At the time where we where developing this result, we where not aware of the article “Regular Model Checking Using Interference of Regular Languages” by Habermehl and Vojnar [13], in which a short remark appear about the relationship between length preserving transducers and non-length preserving transducers. It includes the sketch of an approach similar to ours. Nevertheless the article does not present any proof of this claim.

Our work on this proof is done independently and without any knowledge of the work in [13].

3.4 Summary

In this chapter we have given three results concerning regular model checking.

Most literature in the area of RMC assumes that in general *RMC* is undecidable. However we where not aware of any written proof of this claim, and for completeness we have provided this.

When doing RMC we mainly look at parametric or infinite state-space systems. However we have shown that we gain decidability, if we consider finite initial sets, in fact the problem becomes PSPACE-complete.

The last result is by far the most interesting. Several RMC encodings use modelling tricks similar to the one we use in this reduction — just take the proof regarding undecidability of RMC. Even though this modelling trick is used in many encodings and mentioned for the general case as a remark in [13], we have no knowledge of any formal description and proof of the method.

As mentioned; the construction, regarding the relationship of length- and non-length preserving transducers, only guarantees to preserve the possibility of verifying safety properties.

When we would like to verify liveness properties in a given length preserving RMC-framework, we ask questions like: given an initial set I, a transducer T and a property set X, is it the case that for all $s_0 \in I$ and all translations $c = s_0[T]s_1[T]s_2[T] \cdots [T]s_k[T] \cdots$ that there exists a string $s_k \in X$ that is included in all c? Since I is possibly infinite, we can not check this for every single string in I, but we can search for a counterexample. Then we would ask questions like: given an initial set I, a transducer T and a property set X is it the case that there exists a string $s_0 \in I$ and a computation $c = s_0[T]s_1[T]s_2[T] \cdots [T]s_k[T] \cdots [T]s_k[T] \cdots$, where we revisit a string s_k, and the set of strings computed until the revisit of s_k are disjoint from X ($s_i \notin X$)? Such a lasso-like computation, as illustrated in Figure 3.4, will be a counterexample.

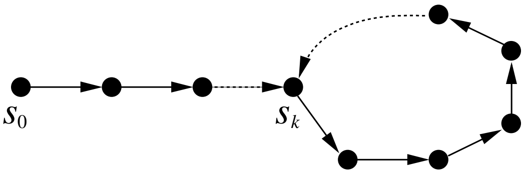


Figure 3.9: Lassoing computation

One method, presented to us by T. Vojnar by email correspondence, is to detect these lassos as follows: instead of a given transducer T it can be modified to T' such that T' has a three nondeterministic branches: (1) a copy of T. (2) a part that copies the input string s into $s\#s'$, s.t. $s[T]s'$, and (3) a transducer part equivalent to T, which works on the string after $\#$. All the three parts of T' can only translate if $s, s' \notin X$. With this new transducer we ask if we can reach a string of the form $w\#w$ for some string w, meaning we found a lasso.

This works for length preserving transducers and, as we claim, for a subclass of non-length preserving ones. If we have a non-length preserving transducer where some translation sequence revisits some states in the transducer and grows the string, e.g. $q_1 \xrightarrow{\alpha_1/\alpha_1'} q_2 \xrightarrow{\alpha_2/\alpha_2'} \cdots \xrightarrow{\alpha_{n-1}/\alpha_{n-1}'} q_n$, where $q_n = q_1$ and $|\alpha_1 \cdots \alpha_{n-1}| < |\alpha_1' \cdots \alpha_{n-1}'|$, then the liveness verification might never end because there is no guarantee that a lasso-like counterexample can be found, because the translation may repeat it self.

However when we have non-length preserving transducers without such loops as above i.e. that only grow every string with a certain constant, then the described method is still theoretically possible.

We suggest using the approach similar to the following, to detect if a transducer preforms this growth in the transducer. We simply reduce the problem to a Bellman-Ford shortest path problem. The Bellman-Ford algorithm can do shortest path analysis of possibly negative weighted graph - and it detects if an arbitrary negative or positive path can be found in the graph. The idea we present is to give weights to the transitions of a given transducer so that:

- -1 is assigned if the given transition is growing, $q \xrightarrow{\varepsilon/a'} q'$,

- 1 is assigned if the given transition is shrinking, $q \xrightarrow{a/\varepsilon} q'$, and

- 0 is assigned if the given transition does not change the length, $q \xrightarrow{a/a'} q'$.

Now the Bellman-Ford shortest path algorithm can check if our transducer grows strings to arbitrary sizes, simply by testing whether there is a negative cycle in the graph or not.

Chapter 4

Cellular Automaton

In this chapter we will look at an interesting model of states evolution systems with local communication, called a Cellular automaton (CA). We shall formally define this model and give some interesting examples.

A (CA) is a simple model that is used to describe a state evolution system with only local communication. The notion of CA was first used by von Neumann [21], to model and reason about self-replicating systems. Today CA have a variety of uses e.g. for election algorithms, artificial life (”Game of Life”), evolutions of bacteria cultures, and traffic flow. Common to these models are the element of local communication.

A CA consists of two parts, a cellular space and a transition rule. The cellular space is a collection of connected cells, where each of these cell is in some state. The cellular space can be multi-dimensional. The transition function defines an update signal, that simultaneously updates the states of all cells. The update of a cell is based on the states of its surrounding cells, called the neighbourhood.

In the 1970’s a CA named Game of Life became very widely known. The Game of Life, is a two dimensional, two state CA. In Game of Life cells can either be alive or dead. A given cell can either die or be born/reborn based on the cells just next to it [29].

We were originally inspired to have a look at CA, when we encountered the article “SAT-based Analysis of Cellular Automata” by D’Antonio and Delzanno [11]. The article presents a way to do bounded reachability analysis of CA by SAT-solving. The article presents how CA can be encoded as a SAT problem, and shows how to analyse these encodings using the SAT solver zChaff [23]. In our work we want to investigate whether we can go beyond bounded model checking, and preform parametric verification of CA, uring RMC.

When analysing CA we are interested in reachability properties (whether a configuration is reachable from an initial configuration), and inverse reachability (given some configuration, can we "backtrack" the evolution so that we will reach the initial configuration).

Our motivation is to examine if RMC is suitable for verification of CA. We do not know about any other work, doing verification of CA using RMC. Most other work conserning CA verification, seams to be focusing on simulation of the CA [11].

4.1 Definitions

In this section we give a formal definition of CA, and define some interesting restricted CA.

Cellular Automaton

Definition 4.1. A *cellular automaton* (CA) is a 4-tuple $C = (d, S, N, \delta)$ where

- $d \in \mathbb{N}$ is the dimension of the cellular space ($\mathbb{Z}^d$),
- S is the finite set of states,
- $N \subseteq \mathbb{Z}^d$ is the neighbourhood, where $n = |N|$, is the size of the neighbourhood, and
- $\delta : S^{n+1} \rightarrow 2^S \setminus \emptyset$ is the transition function.

The number d is called the dimension of the cellular automaton. We use the term 1D-CA to denote a CA of dimension 1 ($d = 1$).

If $N = \{v_1, v_2, \ldots, v_n\}$, then the neighbourhood for a given cell i is obtained by considering the cells $\{i, i + v_1, i + v_2, \ldots, i + v_n\}$. We implicitly assume a fixed ordering of the neighbour cells.

A configuration of CA $C = (d, S, N, \delta)$ is a function $c : \mathbb{Z}^d \rightarrow S$ that assigns a state to each cell of the CA. The set of all configurations is denoted ζ.

All cells in the CA update there states at the same time, based on the states of the neighbourhood.

The global evolution function of a cellular automaton C, $G_C : \zeta \rightarrow 2^\zeta$, is defined by:

$$G_C(c)(i) \in \delta(c(i), c(i + v_1), \ldots, c(i + v_n)) \text{ for any } c \in \zeta, i \in \mathbb{Z}^d.$$

#	Rule
1	nnn → n
2	nnt → n
3	ntn → n
4	ntt → n
5	tnn → t
6	tnt → n
7	ttn → n
8	ttt → n

Table 4.1: Transition function for CA C.

Given an initial configuration c_0, an evolution of c_0 in C is an infinite sequence, $c_0, c_1, \ldots$ such that $c_{t+1} \in G_C(c_t)$. By $Ev^C(c_0)$ we denote all possible evolutions of c_0 in CA C.

A configuration c' is reachable in k steps from configuration c_0, written $c_0 \xrightarrow{k}_C c'$, if there is an evolution $c_0, c_1, c_2, \ldots, c_k, \ldots$ in $Ev^C(c_0)$, such that $c' = c_k$, we write this $c_0 \xrightarrow{k} c'$ if C is clear from the context.

For 1D-CA with neighbourhood $N = \{+1, -1\}$ we write the rules of the transition function as $abc \to b'$, here a is the left neighbour, b is the cell being updated and c is the right neighbour, and b' is the new state of cell b.

◊

Before we defined a CA to be nondeterministic, however a more classical definition of the CA, is where evolution of the CA is deterministic.

Deterministic Cellular Automaton

Definition 4.2. A deterministic Cellular Automaton (dCA) $C = (d, S, N, \delta)$ is defined as in Definition 4.1 but where $|\delta(a_0 \ldots, a_n)| = 1$ for all $a_0, \ldots, a_n \in S$. ◊

Example 4.3. Consider the CA $C = (1, \{t, n\}, \{+1, -1\}, \delta)$ where δ is given by the rules in Figure 4.1. This is a example of a dCA. The CA implements a simple token parsing algorithm, as from the Example in Section 2.2, where a token (T) is passed on the right neighbour, but with the exception that we in this Example have an infinite numbers of processes.

Given a configuration

$$c_0(i) = \begin{cases} t & \text{if} \quad i = 0 \\ n & \quad \text{otherwise} \end{cases}$$

the evolution of this configuration is shown in Figure 4.2. Each line is a configuration that can be reached from the above configuration.

step	...	ℓ_{-2}	ℓ_{-1}	ℓ_0	ℓ_1	ℓ_2	ℓ_3	...
1	...n	n	n	t	n	n	n	n...
2	...n	n	n	n	t	n	n	n...
3	...n	n	n	n	n	t	n	n...
4	...n	n	n	n	n	n	t	n...
$\vdots$	$\iddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 4.2: Evolution of configuration c_0 in CA C

We see that from step one the cell ℓ_0 evolves from state t to state n. The neighbours for cell ℓ_0 is the cells ℓ_{-1} and ℓ_1. As the neighbours has the state n, and cell ℓ_0 has the state t, the update is done according to rule number 3. We see that cell ℓ_1 evolves from state n to state t based on rule number 5, and that all other cells keeps the same state by rule number 1.

In the evolution of c_0 in C, it is clear that we will never use the rules $4, 6, 7, 8$, from now one we will only list the rules that are necessary for the evolution.

△

We will now look at a special kind of 1D-CA where the cellular space is connected, forming a ring topology.

Circular Cellular Automata

Definition 4.4. A Circular Cellular Automaton (cCA) *of size* $z \in \mathbb{N}$ is defined as a non-deterministic 1D-CA $C = (1, S, N, \delta_C)$, but where:

- $|N| \leq z$, and
- the neighbourhood is obtained by considering the cells $\{i, (i+v_1)_{\text{mod } z}, (i+v_2)_{\text{mod } z}, \ldots, (i+v_n)_{\text{mod } z}\}$ where $N = \{v_1, \ldots, v_n\}$.

Given a configuration c we only consider the cells used in the computation. We write the configuration c using the notation $c = (c(0), \ldots, c(z-1))$.

◇

Example 4.5. Consider the cCA $C = (1, \{t, n\}, \{+1, -1\}, \delta)$ of size 8, with initial configuration $c_0 = (t, n, n, n, n, n, n, n)$ and where δ is given by the rules in Table 4.3.

This CA implements a simple token passing algorithm, where the token is passed on to the right neighbour and where the ”processes” are connected in a ring topology.

#	Rule
1	nnn → n
2	nnt → n
3	ntn → n
4	tnn → t

Table 4.3: Transition function for circular CA C, this only considers the rules used for the evolution of c_0, all rules of the CA is given in Table 4.1 on page 56.

The evolution of configuration c_0 is shown in Figure 4.1. The neighbourhood of a cell is the right and left cell in the ring. Each layer of the circle corresponds to one configuration in the evolution. We see that after 8 steps we will reach the configuration c_0 again.

△

We can also restrict the cellular space to be finite, we call this a bounded cellular automata.

Bounded Cellular Automata

Definition 4.6. A non-deterministic 1D-CA $C = (1, S, N, \delta)$ with initial configuration c_0 is called $z-$*bounded* or a Bounded Cellular Automata (bCA) if, $z \in \mathbb{N}$ and

- S contains a barricade state $\$ \in S$, that satisfies:
  - $\delta(\$, a_1, \dots, a_n) = \$ \ \forall a_1, \dots, a_n \in S$, and
  - $\delta(a_0, a_1, \dots, a_n) \neq \$ \ \forall a_0 \in S \setminus \{\$\}$, $\forall a_1, \dots, a_n \in S$, and
- the configuration c_0 is z-bounded, satisfying that for all $i \in \mathbb{N}$:
  - $c(i) = \$$ if $i < 0$,
  - $c(i) = \$$ if $i \geq z$ and
  - $c(i) \neq \$$ if $0 \leq i < z$.

◊

When looking at a configuration of a bounded CA, we only consider the part which does not contain barricade symbols. We write a configuration c of a bCA as $c = (c(0), \dots, c(z-1))$.

Consider any z-bounded CA C, with an initial state c_0. All configurations in from $Ev^C(c_0)$ will be z-bounded.

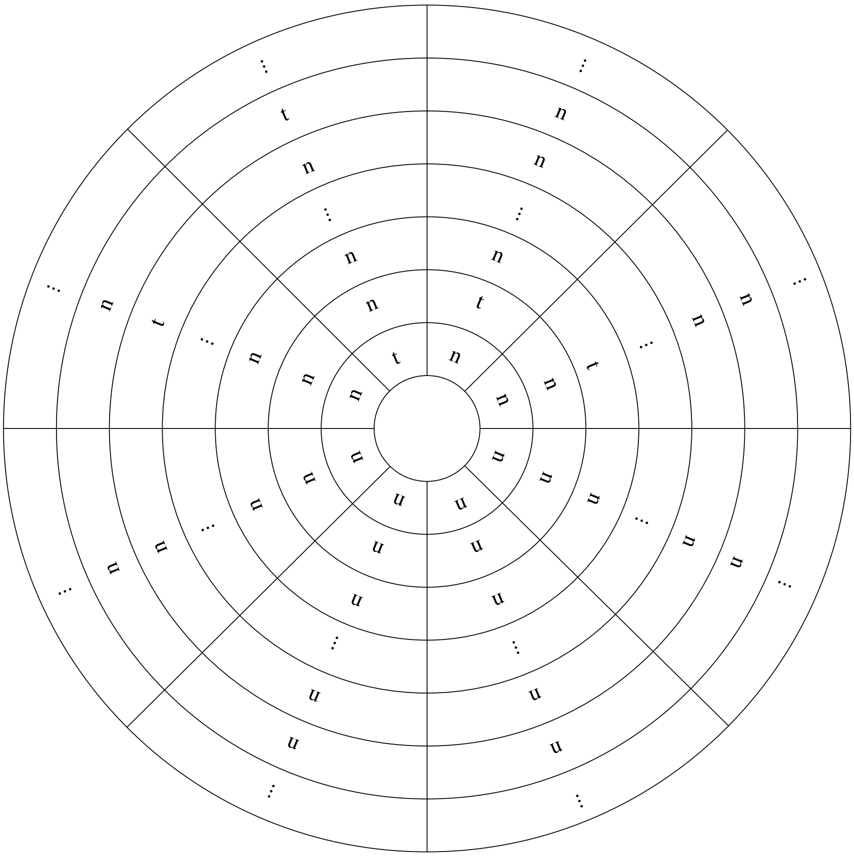


Figure 4.1: The evolution of cCA with initial configuration c_0.

#	Rule
1	nnn → n
2	nnt → n
3	ntn → n
4	tnn → t
5	\$tn → n
6	\$nt → n
7	\$nn → n
8	nn\$ → n
9	tn\$ → t
10	nt\$ → t

Table 4.4: Transition rules for bCA C.

step		l_0	l_1	l_2	l_3	l_4	l_5	
1	. . . \$	t	n	n	n	n	n	\$ . . .
2	. . . \$	n	t	n	n	n	n	\$ . . .
3	. . . \$	n	n	t	n	n	n	\$ . . .
4	. . . \$	n	n	n	t	n	n	\$ . . .
5	. . . \$	n	n	n	n	t	n	\$ . . .
6	. . . \$	n	n	n	n	n	t	\$ . . .
7	. . . \$	n	n	n	n	n	t	\$ . . .
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.5: Evolution of configuration c_0 in bounded CA C

When '\$' represents the barricade state of bounded CA $C = (1, N, S, \delta)$, we will assume implicitly that $\$ \in S$ and that the transition rules

$$\delta(\$, a_1, \ldots, a_n) = \$$$

is included in δ for all $a_1, \ldots, a_n \in S$.

Example 4.7. Consider the bCA $C = (1, \{t, n\}, \{+1, -1\}, \delta)$ with initial configuration $c_0 = (t, n, n, n, n, n, n, n)$ and where δ is given by the rules in Table 4.4.

This CA implements a simple token passing algorithm, where the token is passed on to the right neighbour and the last process keeps the token.

In Table 4.4 we illustrate a subset of the transition rule δ of the bCA $C = (1, \{a, b, \$\}, N = \{-1, +1\}, \delta)$ with initial configuration $c_0 = (t, n, n, n, n, n)$.

The evolution of configuration c in CA C is illustrated in Table 4.5.

△

Decision Problems

We will now formally define the decision problems for reachability analysis in CA.

The general reachability problem for the most general CA variant:

Definition 4.8. ***CA-reach***

Instance: A non-deterministic CA C and configurations c_0 and c.

Question: Is c reachable from c_0 in C?

◊

The decision problem for circular and bounded CA is defines as the decision problems *cCA-reach* and *bCA-reach*.

Definition 4.9. ***cCA-reach***

Instance: A z-circular CA C with configurations c_0 and c.

Question: Is c reachable from c_0 in C?

◊

Definition 4.10. ***bCA-reach***

Instance: A bCA C with configurations c_0 and c.

Question: Is c reachable from c_0 in C?

◊

We define the decision problem *rcCA-reach* as a restricted reachability problem in cCA, with neighborhood $N = \{+1, -1\}$.

Definition 4.11. ***rcCA-reach***

Instance: A z-circular CA $C = (1, S, N, \delta)$ with configurations c_0 and c and $N = \{+1, -1\}$.

Question: Is c reachable from c_0 in C?

◊

We have now formally defined the CA and the reachability question in a CA.

4.2 CA Algorithms

In this section we will have a look at some different CA algorithms that will later be used in the following chapters.

4.2.1 Rule χ

The Rule χ is a class of 256 different deterministic 1D-CA, with only two state, $S = \{0, 1\}$, and with $N = \{+1, -1\}$. By Rule χ where $0 \leq \chi \leq 255$, we mean the CA with the above definition and the rules in Table 4.6, where the sequence $\chi_1\chi_2\chi_3\chi_4\chi_5\chi_6\chi_7\chi_8$ is the binary encoding of the number χ.

Rule χ
1 1 1 $\rightarrow \chi_8$
1 1 0 $\rightarrow \chi_7$
1 0 1 $\rightarrow \chi_6$
1 0 0 $\rightarrow \chi_5$
0 1 1 $\rightarrow \chi_4$
0 1 0 $\rightarrow \chi_3$
0 0 1 $\rightarrow \chi_2$
0 0 0 $\rightarrow \chi_1$

Table 4.6: Template Rules for the Rule χ CA

Example 4.12. An example of this is Rule 30. The binary encoding of the number 30 is "00011110". Which means that for Rule 30 the values for χ_1 to χ_8 are as follows: $\chi_1 = \chi_2 = \chi_3 = \chi_8 = 0$ and $\chi_4 = \chi_5 = \chi_6 = \chi_7 = 1$.

△

Even though this class of CA seems simple, some of them have quite interesting behaviours. E.g. Rule 184 has been used to simulate traffic flow models [20], Rule 30 has been proposed as a random generator and have even been proposed applied in the field of cryptography[30]. Also Rule 110 turn out to have a complex behaviour, in fact Rule 110 has been proved Turing Complete [28], with two notable difference from the Turing machine: the computation continues infinitely, it has no halting state and the initial configuration of the CA is finite, where the initial configuration of a Turing machine, has to be finite.

4.2.2 Firing Squad Synchronisation Problem

The Firing Squad Synchronisation Problem (FSSP) is he problem of synchronisation between a number of processes with local communication. The problem is normally formulated as: a number of soldiers need to fires the canons at a enemy city, to be certain that the attack is successful, then need to fire there canons a exactly the same time. All soldiers can receive a message from the solders to the left and right of him. The FSSP is to find a solution such that all solders firers there cannon simultaneously. We can encode a solution of this problem using a bCA.

There exists a number of solutions to the problem. A smallest five-state solution has been shown by H. Umeo and T. Yanagihara [26]. And a minimum time solution has be show by Waksman [12] using 2n-2 steps, where n is the number of solders. The website [19] gives the CA rules for different solutions to the problem.

We consider Jacques Mazoyer's [16] six-state minimal solution to the FSSP. This solution works by finding the middle cell, by sending two messages. One message traveling by speed 1, and one with speed 1/3. When one cells receives both messages at the same time, we know this is the middle cell, dividing the cells into two segments. Now the middle of these two cells is found in a similar way, until a cell reaches the firer state, see Figure 4.2 for an illustration the solution.

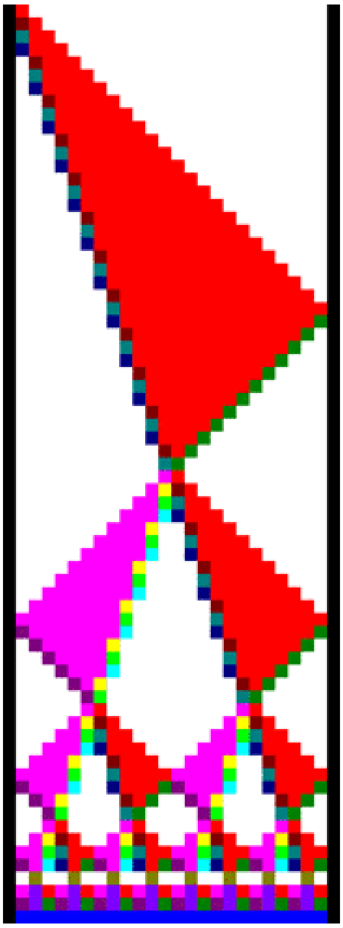


Figure 4.2: Figure of the Jacques Mazoyer's six-state minimal solution to the FSSP. source: wikipedia.org

Chapter 5

Verification of Cellular Automata

In this chapter we will have a look on how verification of CA can be done using RMC. We present two results in this chapter. First we show that bCA and cCA have the same expressiveness up to isomorphism, and then we show a reduction from cCA to RMC.

5.0.3 Circular vs. Bounded Cellular Automata

In this section we will have a look at the expressiveness of the different types of CA.

Definition 5.1. Given CA C and configuration c_0, we define $Conf^C(c_0)$ as the set of all reachable configurations from c_0 in C. ♢

Definition 5.2 (Isomorphism)**.** Given CA C, C' and configurations c_0, c'_0, the pairs (C, c_0), (C', c'_0) are isomorphic, $(C, c_0) \cong_i (C', c'_0)$, iff

1. there is a bijection $f : Conf^C(c_0) \longrightarrow Conf^{C'}(c'_0)$, and
2. $c_1 \to_C c_2 \iff f(c_1) \to_{C'} f(c_2)$ for all the configurations c_1, c_2 of C.

♢

The relationship between the different 1D-CA, with regards to isomorphism, is illustrated in Figure 5.1.

The general CA are of coarse more expressive than the bounded and the circular ones. This is because bCA and cCA configurations are restricted, they only evolve to a finite number of different configuration. In Example 4.3 on page 56 we have an example of a CA with an infinite number of reachable

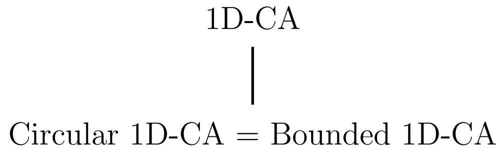


Figure 5.1: Relationship between different types of CA up to isomorphism.

configurations, and of course this can not have an isomorphic bounded or circular CA, making it more expressive than any bCA or cCA.

However bCA and cCA are isomorphic

Theorem 5.3. *For any 1D-bCA there is an isomorphic 1D-cCA, and for any 1D-cCA there is an isomorphic 1D-bCA.*

To prove Theorem 5.3 we will have to prove that circular CA are as expressive as bounded CA up to isomorphism and vice versa. This is formulated in Lemma 5.5 and Lemma 5.6, so by proving these lemmas we prove Theorem 5.3.

First we will prove Lemma 5.5. For the bijective function we will need the *distance* of a given neighbourhood.

Definition 5.4. The *distance* of neighbourhood N in a 1D-CA, denoted $dist(N)$, is defined as:

$$\texttt{max}(\{|v_i| \mid v_i \in N\}).$$

◊

Lemma 5.5. *For any 1D-bCA B and configuration c_0 there exists a 1D-cCA C and a configuration c'_0 such that $(B, c_0) \cong (C, c'_0)$. Constructing C from B is possible in linear time.*

Proof. Assume a given 1D-bCA $B = (1, S_B, N_B, \delta_B)$ of size z with configuration c_0. We now create 1D-cCA $C = (1, S_C, N_C, \delta_C)$ of size z' with configuration c'_0, that will behave like B.

Given

$$c = (a_0, a_1, \ldots, a_{z-1}) \in Conf^B(c_0),$$

we define

$$f(c) = (a_0, a_1, \ldots, a_{z-1}, \overbrace{\$, \$, \ldots, \$}^{dist(N_B)}).$$

Note that this means that the size of C will be $z' = z + dist(N_B)$.

The construction of the cCA is very simple, we simply let $S_C = S_B$, $N_C = N_B$ and $\delta_C = \delta_B$.

It is clear from the construction that for each configuration in $Conf^B(c_0)$ we will get exactly one configuration in $Conf^C(f(c_0))$, making it a bijection. Because we added $dist(N_B)$ '\$' in the end of every configuration, we get that the outer cells in configurations c_0 and c_0' has exaclty the same neigbourhood, and Since $S_C = S_B$, $N_C = N_B$ and $\delta_C = \delta_B$ it is easy to see that

$$c_1 \rightarrow_B c_2 \iff f(c_1) \rightarrow_C f(c_2).$$

Regarding the complexity, the only thing we did was to add some extra cells (in state \$) to the configuration c_0, which can be done in linear time. □

Lemma 5.6. *For any 1D-cCA C and configuration c_0 there exists a 1D-bCA B and a configuration c_0' such that $(C, c_0) \cong (B, c_0')$.*

Proof. Assume a given 1D-cCA $C = (1, S_C, N_C, \delta_C)$ of size z with configuration c_0. We now create 1D-bCA $B = (1, S_B, N_B, \delta_B)$ of size one with configuration c_0', that will behave like C.

The idea is to make a state in S_B for all different configurations in $Conf^C(c_0)$. Note that by doing so, we reduce the configurations (of possibly many cells) in $Conf^C(c_0)$ to a single cell representing an entire configuration from $Conf^C(c_0)$.

Given configuration $c = (a_0, a_1, \ldots, a_{z-1})$, we define a one-state representation of c to be $\langle c \rangle = a_0 a_1 \cdots a_{z-1}$.

Given $c = (a_0, a_1, \ldots, a_{z-1}) \in Conf^C(c_0)$, we define $f(c) = (\langle c \rangle)$, that is a configuration of only one state $\langle c \rangle$.

We construct the bounded CA $B = (1, S_B, N_B, \delta_B)$, so that $N_B = \emptyset$, $S_B = \big\{ \langle c \rangle \mid \text{for all } c \in Conf^C(c_0) \big\}$, and for all $c \rightarrow_C c'$, we add a rule $\delta_B(\langle c \rangle) = \langle c' \rangle$, we also add the rule $\delta_B(\$) = \$$.

From this construction it is clear that for each configuration in $Conf^C(c_0)$ we will get exactly one configuration in $Conf^B(f(c_0))$, because the only thing f^C does, is reduce a configuration to a single state for every configuration in $Conf^C(c_0)$. This defines the required bijection.

Since 1D-bCA B is build to mimic the exact behaviour of 1D-cCA C, it is clear that from this construction we have that

$$c_1 \rightarrow_C c_2 \iff f(c_1) \rightarrow_B f(c_2).$$

□

Note that the running time of the reduction from cCA to bCA is exponential in the size, z, of cCA C, the running time of the reduction is $O(|Conf^C(c_0)|^2) = O(|S_C|^{2z})$.

5.1 Circular Cellular Automata Verification using RMC

Recall Definition 4.11 on page 61 of the restrict decision problem of cCA analysis *rcCA-reach*. The problem is if a configuration is reachable in a 1D-cCA with neighbourhood $N = \{+1, -1\}$.

In this section we will present a reduction from *rcCA-reach* to *RMC*. In Section 5.0.3 we showed how to convert a bCA to a cCA in linear time, so the following reduction also works for bCA.

Later we will consider how to expand this to greater neighbourhoods and optimise the reduction for bCA.

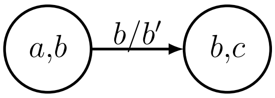
5.1.1 Reduction

Given a circular CA $C = (1, S_C, N_C, \delta_C)$ with neighbourhood $N_C = \{-1, +1\}$ and configurations c_0 and c', we construct an RMC-framework $R = (I, T, B)$ such that for every $k \in \mathbb{N}_0$

$$c_0 \xrightarrow{k} c' \iff I \circ T^k \cap B \neq \emptyset \ .$$

Idea In CA the evolution of one cell depends on its neighbourhood. Given a cell i of CA $C = (1, S_C, N_C, \delta_C)$ with $N_C = \{-1, +1\}$, we consider the cells just to the left and just to the right of the cell i to determine the next state of cell i. In a transducer we only consider one symbol to decide what the output symbol will be. However the output symbol also depends on the current state of the transducer. To model a bCA C we will use the states of the transducer to remember the neighbourhood of the CA. We will name the states of the transducer (x, y) where x is the symbol just read, and y is the next symbol we need to read. We call the state name (x, y), as seen 'x' and read 'y'. This means that when we construct the transducer, only $\xrightarrow{x/\alpha}$ transitions can be going to the state (x, y) and only $\xrightarrow{y/\alpha'}$ transitions can be going from the state (x, y), where $\alpha, \alpha' \in \Sigma$.

Example 5.7. To give an example of this, consider the rule $abc \to b'$ in a 1D-bCA. We model this rule by the transition $(a, b) \xrightarrow{b/b'} (b, c)$ in the transducer. This transducer part is illustrated as follows:



△

Now as we consider cCA, we will also have to make sure that the first and the last symbol is translated according to its neighbour in the opposite end of the string. As we do not know the value of the last symbol, when we have to alter the first symbol, we have to make a nondeterministic guess the last symbol. We know that we have to use one of the rules in the cCA, so we can make a guess to which rule that fits. Then, when we alter the last symbol, we will have to remember the first encountered symbol, to what symbol we guessed was the last. It is okay to end the computation, if we are in a state where we have seen the symbol we guessed to be the last, and if we are going to read the symbol we remembered as the first.

For this we extend the naming schema from above so that instead of naming states of the transducer (x, y) we name them (f, ℓ, x, y), where f is the first symbol and ℓ is the last symbol in the string. The state (f, ℓ, x, y), can be interpreted as: the first symbol is 'f', guessed that the last symbol is going to be 'ℓ', and just seen 'x' and read 'y' (just as above). This way we remember the first symbol and guess the last symbol. The final transducer will nondeterministically choose a part of the transducer that fits the string.

In the reduction we write $\langle c \rangle = c(0)c(1)\cdots c(z-1)$ to denote the string representation of the configuration $c = (c(0), c(1), \ldots, c(z-1))$.

The algorithm for the reduction is presented in Algorithm 5.

Algorithm 5: Reduction Algorithm from cCA to RMC.

```
Input: A cCA $C = (1, S_C, N_C, \delta_C)$ and configurations $c_0$ and $c'$.
Output: RMC-finite-framework $R = (I, T = (Q_T, \Sigma_T, \delta_T, q_0, F_T), B)$.
begin
    $I := \{\langle c_0 \rangle\}$;
    $B := \{\langle c' \rangle\}$;
    $Q_T := (S \times S \times S \times S) \cup \{start\}$;
    $q_0 := start$ ;
    $F_T := \{(f, \ell, \ell, f) | \ell, f \in S\}$ ;
    $\Sigma_T := S_C$ ;
    forall $abc \to b' \in \delta_C$ do
        add to $\delta_T$ the transitions: $(f, \ell, a, b) \xrightarrow{b/b'} (f, \ell, b, c)$ for all
        $f, \ell \in S$ ;
        add to $\delta_T$ the transition: $start \xrightarrow{b/b'} (b, a, b, c)$ ;
    return $R$
end
```

In the reduction we create a transducer with at most $O(|S|^4)$ states, and $O(|\delta_C| \cdot |S|^2)$ transitions. This makes the reduction polynomial.

Example 5.8. As an example of the reduction, let us have a look at the token passing ring example, from Example 4.5 on page 57.

Recall the definition of the CA $C = (1, S_C = \{t, n, \}, N_C = \{-1, +1\}, \delta_C)$, where the significant rules are $tnn \to t, ntn \to n, nnt \to n, nnn \to n$. The illustration in Figure 5.2 is the transducer resulting from our reduction.

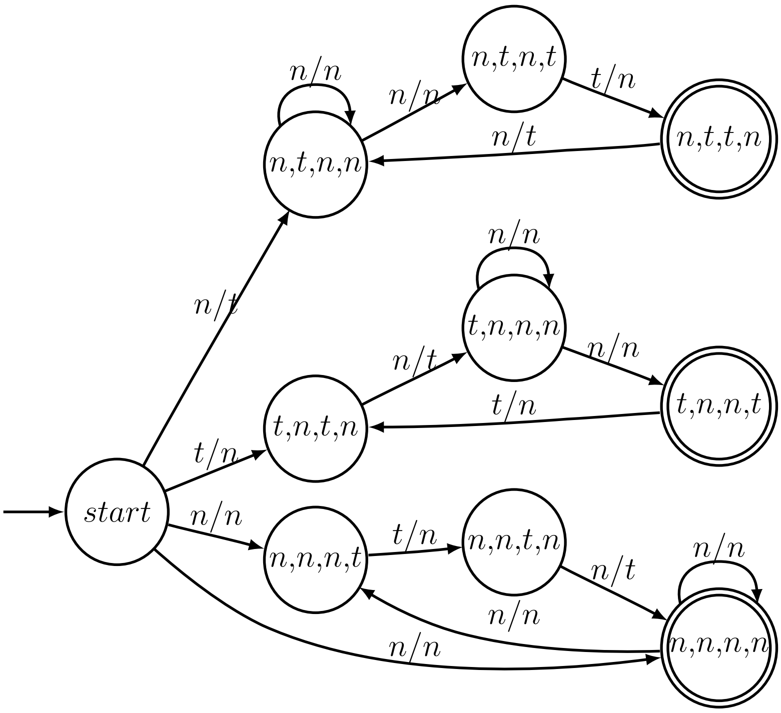


Figure 5.2: Transducer created by the Algorithm 5 on the cCA that implements a token passing ring.

We can see that by following the transducer on the string *nnnt*, we get the translation $start \xrightarrow{n/t} (n, t, n, n) \xrightarrow{n/n} (n, t, n, n) \xrightarrow{n/n} (n, t, n, t) \xrightarrow{t/n} (n, t, t, n)$ - leading to the string *tnnn*, which is translated to *ntnn*, which is translated to *nntn*, which again is translated to *nnnt*, just like C would have done.

$\triangle$

5.1.2 Correctness of Reduction

Theorem 5.9. *Given a cCA $C = (1, S_C, N_C, \delta_C)$ of size z and configurations c_0 and c', and given a RMC-framework, $R = (I, T, B)$ constructed according to Algorithm 5, we have*

$$c_0 \xrightarrow{k} c' \iff I \circ T^k \cap B \neq \emptyset.$$

To prove Theorem 5.9 we expand the configurations, such that we get the following claim:

Claim. *Given a cCA $C = (1, S_C, N_C, \delta_C)$ of size z and configurations $c_0 = (a_0, a_1, a_2, \ldots, a_m)$ and $c' = (b_0, b_1, b_2, \ldots, b_m)$ (where $m = z - 1$), and given a RMC-framework, $R = (I, T, B)$ constructed according to Algorithm 5, we have*

$$(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k} (b_0, b_1, b_2, \ldots, b_m)$$

$$\iff$$

$$b_0 b_1 b_2 \cdots b_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^k.$$

Proof. By induction on k.

"$\Longrightarrow$-direction"

Basis step $(k = 0)$

We have to show that if $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{0} (b_0, b_1, b_2, \ldots, b_m)$, then $b_0 b_1 b_2 \cdots b_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^0$.

As $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{0} (b_0, b_1, b_2, \ldots, b_m) = (a_0, a_1, a_2, \ldots, a_m)$, we have to show that $a_0 a_1 a_2 \cdots a_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^0$.

Since $\{a_0 a_1 a_2 \cdots a_m\} \circ T^0 = \{a_0 a_1 a_2 \cdots a_m\}$, we have shown the basic step.

Induction Hypothesis (k)

$$\begin{aligned} \text{IH(k)} \equiv (a_0, a_1, a_2, \ldots, a_m) &\xrightarrow{k} (b_0, b_1, b_2, \ldots, b_m) \\ &\Longrightarrow b_0 b_1 b_2 \cdots b_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^k. \end{aligned}$$

Induction step $(k + 1)$

We have to show that if $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k+1} (d_0, d_1, d_2, \ldots, d_m)$ then $d_0 d_1 d_2 \cdots d_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^{k+1}$.

We know that if $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k+1} (d_0, d_1, d_2, \ldots, d_m)$ then $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k} (b_0, b_1, b_2, \ldots, b_m) \xrightarrow{1} (d_0, d_1, d_2, \ldots, d_m)$. Now by IH($k$) we have that $b_0 b_1 b_2 \cdots b_m \in \{a_0 a_1 a_2 \cdots a_m\} \circ T^k$. So it is enough to show that $b_0 b_1 b_2 \ldots b_m [T] d_0 d_1 d_2 \ldots d_m$.

We know that when cCA C evolves from $(b_0, b_1, b_2, \ldots, b_m)$ to $(d_0, d_1, d_2, \ldots, d_m)$ then for each cell i, where $0 \leq i \leq m$ and $b_{-1} = b_m$, there is a rule $b_{i-1}b_ib_{i+1} \rightarrow d_i$ updating the cell i. Because of the way we construct T, we will for each rule that fits this pattern have an equivalent part in the transducer.

Since $b_{i-1}b_ib_{i+1} \rightarrow d_i$ will have transducer parts that are sequentially connected (illustrated in Example 5.8), and since the accepting states are states that fit (f, ℓ, x, y) where $x = \ell = s_{-1} = b_m$ and $y = f = s_0 = b_0$, we will, when C evolves from $(b_0, b_1, b_2, \ldots, b_m)$ to $(d_0, d_1, d_2, \ldots, d_m)$, get a translation from $b_0b_1b_2 \cdots b_m$ to $d_0d_1d_2 \cdots d_m$ by T.

This means that we can translate $b_0b_1b_2 \cdots b_m$, by using the following translation $b_0b_1b_2 \cdots b_m[T]d_0d_1d_2 \cdots d_m$: $\mathit{start} \xrightarrow{b_0/d_0} (b_0, b_m, b_0, b_1) \xrightarrow{b_1/d_1} (b_0, b_m, b_1, b_2) \xrightarrow{b_2/d_2} (b_0, b_m, b_2, b_3) \xrightarrow{b_3/d_3} (b_0, b_m, b_3, b_4) \xrightarrow{b_4/d_4} \ldots \xrightarrow{b_{m-1}/d_{m-1}} (b_0, b_m, b_{m-1}, b_m) \xrightarrow{b_m/d_m} (b_0, b_m, b_m, b_0)$. As the state (b_0, b_m, b_m, b_0) is accepting we have given a translation, showing the induction step.

"$\Longleftarrow$-direction"

Basis step $(k = 0)$
We have to show that if $a_0a_1a_2 \cdots a_m \in \{a_0a_1a_2 \cdots a_m\} \circ T^0$ then $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{0} (a_0, a_1, a_2, \ldots, a_m)$.

Since $\{a_0a_1a_2 \cdots a_m\} \circ T^0 = \{a_0a_1a_2 \cdots a_m\}$ then of coarse $a_0a_1a_2 \cdots a_m \in \{a_0a_1a_2 \cdots a_m\}$. And since $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{0} (a_0, a_1, a_2, \ldots, a_m)$ we have shown the base case.

Induction Hypothesis (k)

$$\mathrm{IH}(k) \equiv b_0b_1b_2 \cdots b_m \in \{a_0a_1a_2 \cdots a_m\} \circ T^k$$
$$\Longrightarrow (a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k} (b_0, b_1, b_2, \ldots, b_m).$$

Induction step $(k + 1)$
We have to show that if $d_0d_1d_2 \cdots d_m \in \{a_0a_1a_2 \cdots a_m\} \circ T^{k+1}$ then $(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k+1} (d_0, d_1, d_2, \ldots, d_m)$.

If $d_0d_1d_2 \cdots d_m \in \{a_0a_1a_2 \cdots a_m\} \circ T^{k+1}$ then there is a string $b_0b_1b_2 \cdots b_m \in \{a_0a_1a_2 \cdots a_m\} \circ T^k$, s.t. $b_0b_1b_2 \cdots b_m[T]d_0d_1d_2 \cdots d_m$.

Now by applying IH(k) we know that:

$$(a_0, a_1, a_2, \ldots, a_m) \xrightarrow{k} (b_0, b_1, b_2, \ldots, b_m).$$

So we need to show that $(b_0, b_1, b_2, \ldots, b_m) \xrightarrow{1} (d_0, d_1, d_2, \ldots, d_m)$.

We know that as T translates $b_0 b_1 b_2 \cdots b_m$ to $d_0 d_1 d_2 \cdots d_m$ then there is a translating computation:
$start \xrightarrow{b_0/d_0} (b_0, b_m, b_0, b_1) \xrightarrow{b_1/d_1} (b_0, b_m, b_1, b_2) \xrightarrow{b_2/d_2} (b_0, b_m, b_2, b_3) \xrightarrow{b_3/d_3} (b_0, b_m, b_3, b_4) \xrightarrow{b_4/d_4} \ldots \xrightarrow{b_{m-1}/d_{m-1}} (b_0, b_m, b_{m-1}, b_m) \xrightarrow{b_m/d_m} (b_0, b_m, b_m, b_0)$, where $(b_0, b_m, b_m, b_0) \in F$.

Each transition in the translation, corresponds to one cell in the CA gets state updated. Now we can find a computation in the CA by looking at the above computation where $1 : b_m b_0 b_1 \rightarrow d_0$, $2 : b_1 b_2 b_3 \rightarrow d_1$, …, $m : b_{m-1} b_m b_0 \rightarrow d_0$.

Now by applying the above rule i to cell i, we get an evolution step $(b_0, b_1, b_2, \ldots, b_m) \xrightarrow{1} (d_0, d_1, d_2, \ldots, d_m)$, which shows the induction step.

□

This shows the claim, and thereby also Theorem 5.9.

5.1.3 Greater Neighbourhoods

The reduction from cCA to RMC could easily be extended to greater neighbourhoods. To simplify the proof and reduction idea we have left this out of the formal description.

The extension can be done by remembering more values in a states of the transducer, so that for neighbourhood $N = \{+2, +1, -1, -2\}$ we will name the states $(f, l, n_1, n_2, n_{-1}, n_{-2})$, where n_1, n_2 is the values of the last read symbols, and n_{-1}, n_{-2} is the values we guess will be seen. Recall Definition 5.4 of the distance of a neighbourhood, then we see that the size of the reduction grows exponentially in the distance of the neighbourhood.

5.1.4 Optimisation for bCA

We have already shown how a bCA can be converted to a cCA, meaning we can use the cCA reduction for a bCA. However the reduction from bCA to RMC can be simplified. If we have a bCA, the reduction becomes even simpler. We do not need to remember and guess the start and end symbol, we only need to remember the neighbourhood in the states, which reduces the number of states in the transducer considerately.

We also simplify the start and termination criteria by creating a special '*end*'-state, which has an in-going transition $\xrightarrow{\$/\$}$ from all states, which is going to check for the barricade symbol '\$'. Likewise we create a special ‘*start*’-state, which has $\xrightarrow{\$/\$}$ going to every state that has seen \$. The simpler reduction for bCA is presented in Algorithm 6.

Algorithm 6: Reduction Algorithm from bCA to RMC.

```
Input: A cCA $C = (1, S_C, N_C, \delta_C)$ and configurations $c_0$ and $c'$.
Output: RMC-finite-framework $R = (I, T = (Q_T, \Sigma_T, \delta_T, q_0, F_T), B)$.
begin
    $I := \{\langle c_0 \rangle\}$;
    $B := \{\langle c' \rangle\}$;
    $Q_T := (S \times S) \cup \{start, end\}$;
    $q_0 := start$ ;
    $F_T := \{end\}$ ;
    $\Sigma_T := S_C$ ;
    forall $abc \to b' \in \delta_C$ do
        add to $\delta_T$ the transitions: $(a, b) \xrightarrow{b/b'} (b, c)$ ;
    forall states $(\$, y) \in Q_T$ do
        add to $\delta_T$ the transitions: $start \xrightarrow{\$/\$} (\$, y)$ ;
    forall states $(x, \$) \in Q_T$ do
        add to $\delta_T$ the transitions: $(x, \$) \xrightarrow{\$/\$} end$ ;
    return R
end
```

This reduction creates a transducer that has at most $O(|S|^2)$ and at most $O(|\delta_C|)$ transitions, making the reduction polynomial.

5.2 Summary

In this chapter we have shown how to transform bCA and cCA into RMC-frameworks. This makes it possible to model check bCA and cCA with RMC. Our approach is as such more general than the approach from ”SAT-Based Analysis of Cellular Automata”[11], as we can model an arbitrary number of evolutions and not just a limited number of evolutions. We also have the possibility of checking a given bCA or cCA with configurations of arbitrary length, though this feature comes with the trade-off of undecidability. In the next chapter we will make a number of experiments to determine the usefulness and effectiveness of verifying CA with RMC.

The results in this Chapter only considers CA with a cellular space of one dimension. We do at this time not have any good reduction for higher dimension CA, where one evolution in the CA corresponds to one appliance of the transducer. In the case of CA with larger dimensions, we would start by investigate a mini-step encoding, where several steps in the transducer, corresponds to one step in the CA.

Chapter 6

Experiments

In this chapter we will look at a number of experiments for verification of CA using RMC. We have made a prototype implementation of the reduction algorithm presented in Section 5.1, converting a cCA to RMC. The tool reads a CA file, based on our own file format, and outputs files that can be used directly for RMC verification in the ARMC tool[6].

For more details about the prototype tool, its implementation and the CA encoding of the experiments, consult Appendix A. The test computer is a 3000+ AMD Sempron with 1GB of RAM. We have not used any of the optimizations, or different settings in the tool to optimize the verification time.

All experiments is based on a CA algorithm on which we have used our tool to create transducer for the RMC. In the definition of reachability in CA, we defined it as, if from one configuration can reach an other. However in the experiments we are not only interested in weather one configuration can reach an other, we are also interested, in the more difficult question, whether for a possible infinite set of configuration, can we, by the evolution in the CA reach an other, possible infinite, set of infinite configurations?

6.1 Token Passing Protocol

As a proof of concept experiment we have used the Token Passing protocols from Example 4.7 on page 60, and Example 4.5 on page 57. We have created the bCA- and cCA-RMC-framework with our tool. Notice that we used the optimised bCA reduction to construct the transducer for the bCA. For comparison we have also made an encoding of the protocols by hand, with the aim of maximum efficiency.

We have for all versions tested the property: “is there always exactly one token present at any time?" We have performed the verification for an unbound number of processes, where the first process holds the token.

The results of the experiments are illustrated in Table 6.1. The column bCA illustrate test data for the bCA version, and the column bRMC illustrate the test data for the "hand" encoding of the bounded CA. Similar for the columns cCA and cRMC only they are data for the circular token passing protocol. The row "time" shows the CPU running time of the verification in seconds. The row "states" shows the number of states in the transducer and the row "transitions" show the number of transitions in the transducers.

	bCA	bRMC	cCA	cRMC
states	9	3	10	6
transitions	16	4	20	9
time	<0.01	< 0.01	< 0.01	< 0.01

Table 6.1: Experiments for bounded and circular token passing

Our experiments show that all models verify in under 1 second. However the reduction is a little less efficient in the number of states and transitions, but in this case not considerably larger.

There is a difference in the degree of difficulty for modelling the algorithms. It is our experience that the CA encoding, especially in the case of the circular token passing protocol, is consideratetly easier and more natural, than by encoding the transducer by hand.

Next we will verify some CA which have a more complex behaviour than the Token Passing example.

6.2 Rule χ

Recall the Rule χ group of CA, introduced in Section 4.2.1. We have used Rule 30 and Rule 184 to experiment with the efficiency of our automatic conversion. We have made a bCA implementation of the rules, details about this can be found in Appendix A.2.

When analysing the group of Rule χ CA, we are interested if a special pattern can be reached by the CA. In the case of Rule 184, which can be used to model traffic flow of cars, it might be that we want to know if we from some configuration can reach a configuration where there is a queue with more than 10 cars. We have chosen Rule 30 and Rule 184, because Rule 30 is known as one of the more complex CA in Rule χ, and Rule 184 because it has an obvious and initiative use case.

n	1^n	1^n0^n	$(10)^n$
2	< 0.01	< 0.01	<0.05
3	<0.01	<0.01	0.10
4	0.1	0.26	0.41
5	0.14	0.3	0.41
6	0.48	4.77	0.76
7	0.46	4.69	1.50
8	0.60	4.94	2.05
9	14.61	> 5m	2.36
10	14.57	-	7.15
11	> 5m	-	7.39
12	-	-	26.1
13	-	-	25.4
14	-	-	129
15	-	-	120
16	-	-	489
17	-	-	>5m

Table 6.2: Experimental results for Rule 30, n specify the length of the pattern, and the values is the verification in seconds.

For Rule 30 we have tested, how large patterns of the forms "1^n", "$(10)^n$" and "1^n0^n" we can reach. The initial set of the test is described by the regular expression 0^*10^*. The results for Rule 30 are displayed in Table 6.2. The tests we ran all verified. The tests marked with '-' means that the instance has not been tested. The tested sequences are displayed in the columns and each row shows the length of the sequences. The running time of the verification is written in used seconds of CPU time.

We see that reaching configurations that include pattern 1^n is possible within a reasonable time limit is possible for $n \leq 10$. When comparing the results for the pattern $(10)^n$ with the pattern 1^n, we see that the time used to reach configurations that include the pattern $(10)^n$, where $n < 9$, is larger than the time used to reach configurations that include the pattern 1^n. However, for $9 < n < 17$ the configurations with pattern $(10)^n$ is found faster than those that include 1^n. This is even though the pattern is of the double size. For the pattern "1^n0^n", we can verify the existence of the pattern op to $n = 8$. The reason for the faster verification time for the sequences of $(10)^n$ is that the over abstraction works better for this pattern.

For Rule 184 we have tested how long a queue (1^n) we can get and how long a sequence of cars that has exactly sufficient space between each other

n	1^n	$(10)^n$
2	B 0.02	B 0.03
3	B 0.04	B 0.05
4	B 0.16	B 0.03
5	B 0.15	B 0.35
6	H 0.12	B 2.00
7	H 0.08	B 3.39
8	H 0.1	B 3.64
9	H 0.12	B 4.31
10	H 0.13	B 4.54

Table 6.3: Experiments results for Rule 184, n specify the length of the pattern, and the values is the verification in seconds and the answer (B)roken or (H)olds.

to continue forward ($(10)^n$) we can get. For these experiments we used the set of initial configurations that is described by

$$0^*111011110010100101011100000000011101010010^* .$$

The results for Rule 184 are displayed in Table 6.3. The column shows the tested sequence, and each row shows how long sequences are. The values of the running time for the verification is presented in seconds of CPU time.

The experiments show that from the specified set of initial configurations we can reach a queue of 6 cars. The pattern of "$(10)^n$" is reachable up to at least $n = 10$. The verification time for these problems are all under 5 seconds.

6.3 Firing Squad Synchronisation Problem

Recall the firing squad synchronisation problem from Section 4.2.2. We have run the experiments trying to verify The rules of the CA where taken from the website [19].

We tested for the property, "can we reach a configuration where one or more soldiers are shooting and others are not?", for the any number of soldiers.

We tried several different things and settings of the ARMC tool, to try verify the property, but the tool did not terminate. After analysing the problem further we discovered that, because of the nature of the problem, we can not verify it with RMC. For RMC to be possible it must be possible to describe the set of configurations with a regular set. However an important step

n	Time
2	0.03
3	>11m

Table 6.4: The results for verification of the FSSP using a restricted initial set.

n	Time
2	0.02
3	0.04
4	0.09
5	0.11
6	0.18
7	16.1
8	17.5
9	260
10	299

Table 6.5: Experiments results for FSSP, n specify the number of soldiers as solution is found for, and the values is the verification in seconds.

in the FSSP algorithms is to find the soldier in the middle of the configuration, which means the at least the non regular pattern $-^nM-^n$ is included in the closure. If the bad set is indeed intersecting, then in ARMC we will have to refine until we describe every single strings in a given set, which is possibly infinite — hence this is not possible.

However we might still be able to do some bounded model checking of FSSP. The first approach is to restrict the initial to only one string of some length. Even though we might expects a quick answer, based on the decidability result in Section 3.2, this might not be so. Because of the abstract-refine procedure, we may in fact slow down the process considerably, or even loop. In Table 6.4 the result of the bounded experiments are presented. The results shows that we can verify this for up to only two soldiers, with a reasonable time limit.

In the last test we performed, we changed the property so that we ask if we can reach a situations where n soldiers fires at the same time. This way we can verify the existence of a solution for up to n soldiers. We let the initial set consist of an arbitrary number of soldiers, and the "bad" set be a solution of n firing soldiers. Table 6.5 presents the results of these tests. By increasing the value n, we see that we can verify the existence of a solution to the FSSP for up to about 10 soldiers, within a reasonable time limit.

6.4 Alternating Bit Protocol

Finally, we have done experiments with the Alternating Bit Protocol (ABP). The details can be found in the Appendix A.3. We modelled the simple ABP as a 1D-bCA. In the CA encoding of the protocol, we made use of nondeterministic CA, to model the unbounded lossy channels and resend actions of the sender. We used the bCA optimised reduction to create the transducer. The ABP CA has 2586 rules, and this gave us a minimised transducer of 64 states, and 1381 transitions.

We provided an infinite number of start configurations, so that the number of cells representing the two-way lossy channel would be of arbitrary size. We checked that the model would preserve the order of the messages transferred by the protocol.

The verification of the model took about 1.69 seconds. The creators of the ARMC tool, had in the tool test-suite an implementation of the ABP. For comparison we have used their designed transducer, which has 26 states and 121 transitions, the verification uses 0.34 seconds of CPU time. Table 6.6 summarise the results.

	bCA	RMC
states	64	26
transitions	1381	121
time (in sec.)	1.69	0.34

Table 6.6: Data for the ABP Experiments.

We see that the CA-version, compared to the "hand made" (RMC) version, is not considerately larger. Neither is the verification time. The problem designing a correct version of the ABP as a transducer, compared to an automatic conversion is however worth noticing, but this comes with the price of three times as many states and ten times as many transitions.

6.5 Summary

The experiments have shown that parametric verification of CA is indeed possible. Our experiments with the ABP, have shown that we can verify a nondeterministic bCA, with 2586 rules, using our reduction and the ARMC tool in under 2 seconds. Also the reduction itself runs in less than 2 seconds. The price of the automatic reduction is a verification time about 5 times slower and a larger transducer than the "hand made" version, but still within reasonable time. However manually creating the transducer is a slow and tedious task, and may introduce mistakes into the model.

From our experiments with the Rule χ, we have shown that our reduction and RMC can be used for behavioural analysis of bCA evolutions. The experiments show that we can verify the existence of patterns with length up to about 20-30. The experiments also show that there can be large differences in when verifying different types of patterns. When verifying the Rule χ, it would have been a great advantage if we were able to do liveness verification.

The experiments with Jacques Mazoyer's [16] six-state minimal solution to FSSP, showed that this kind of CA is not verifiable using this approach. However, by performing reachability for bounded configurations, we were able to verify the existence of a solution to FSSP for up to 10 soldiers, within a reasonable time limit. Unfortunately we were not able to verify that we cannot reach a configuration where a some number of soldiers fire while other do not.

We believe, based on our experiments that RMC is a candidate method to make verification of CA. But before practical usage will be feasible, the verification times will have to be improved. In general, by our experience and knowledge of RMC, we deem that RMC is well suited for some domain specific problems. But we do not think of RMC as a general purpose verification technique. Too many times one would encounter problems with real world problems and algorithms that does not have a state space that can be described by regular languages.

Chapter 7

Conclusion and Future work

7.1 Conclusion

This thesis focuses on the area of regular model checking (RMC). The thesis main contributions are divided into two parts. Part one concerns the expressiveness of RMC, where we provide proof of three claims about RMC. The second part concerns Cellular Automata (CA), the expressiveness of different CA variants, and how verification of these can be done using RMC. This includes a number of experiments.

Considering RMC, in Section 3.1 we provide a proof of the folklore result that RMC is Turing powerful. Even though this result is well known, we have no knowledge of any publications that prove this. We prove that RMC is Turing powerful by giving a reduction from the halt problem of a two-counter Minsky Machines to RMC. To make the reduction we used a well known modelling trick. In Section 3.3, we formulated this modelling trick formally, and prove that no extra power is added to systems modelled with a non-length preserving transducer instead of a length preserving one. The reduction preserves the possibility of verifying safety properties. Unfortunately, there is still no way of guaranteeing that liveness properties can be verified when this modelling trick is used. This is because of the fact that infinitely many different configurations can occur when a given non-length preserving transducer translates a given finite string. We have, in Section 3.4 proposed a method that analyses if a given non-length preserving transducer generates strings of unbounded lengths. If the transducer only grows strings up to a bounded length, or only shortens the strings, then we will be able to use the reduction, and preserve the possibility to verify liveness properties. This extends the previously known applicability of liveness checking in RMC.

In Section 3.2, we have shown that by restricting the initial set so that it is finite, we gain decidability in PSPACE — but we obviously trade this for the ability of verifying parametrised systems. PSPACE-hardness is proven by using a reduction from reachability in one-safe Petri nets to *RMC-finite*. This reduction, also shows how we can encode one-safe Petri nets and that this encoding technique can easily be extended to any bounded Petri net by simply extending the alphabet of the transducers.

The second part of the thesis is concerned with RMC as a method for parametric verification of CA. In particular we are interested in one dimensional bounded (bCA) and circular (cCA) CA.

In Section 5.0.3 we show the obvious fact that bCA can be reduced to cCA. As a part of this result, we have shown a liner time reduction from bCA to cCA. We also show the more surprising result that cCA and bCA are similar up to isomorphism, and show an exponential space reduction from cCA to bCA.

In Section 5.1 we show a polynomial time reduction from cCA to RMC. It is obvious that the reduction can also be used for bCA, because of the result about isomorphism of cCA and bCA. However we in Section 5.1.4 we show a better reduction that can be used for bCA.

We have successfully implemented a tool automating the reduction from cCA and bCA to RMC-frameworks. The tool automatically outputs a transducer file that can be used directly in the ARMC tool (Se Section 2.4). For details about our prototype tool consult Appendix A.

We have tested the reduction tool on a number of experiments in Section 6. The experiments show examples of how parametric verification of CA algorithms can be done. We have successfully verified the token passing protocol, both a bounded and circular version, the CA of Rule 30 and Rule 184 and the Alternating Bit Protocol (ABP) with unbounded communication channels.

The most interesting result is by fare the verification of the ABP. The ABP is a non-deterministic bCA, with 2586 rules. By using the reduction, we could verify the ABP protocol using the ARMC tool, in less than two seconds. When comparing the transducer generated by our tool, with the human engineered transducer, we see that the automatic encoding method has an overhead with regards to the size of transducers. However the running time for the experiments where not considerately larger for the CA versions. The experiments also confirms our belief, that modelling algorithms using CA is often easier than RMC-modelling by hand, and the automatic generation also excludes potential human mistakes. Also often CA that need verification are already given, and so they can easily be inserted into our tool and then verified with the ARMC tool.

In the experiments with verification of a solution to the Firing Squad

Synchronisation Problem (FSSP), we experienced that we where not able to verify this CA. We traced this problem back to the fact that the resulting state-space cannot be described by finite-automata. We were still able to do some bounded model checking of the FSSP and so verify that the CA algorithms for FSSP works for at up to ten solders.

After experimenting with CA and RMC, we found that safety properties often did not describe the most interesting properties of a given CA. Often we are interested in if a given property holds for all configurations described by an initial regular set. Which is characterised as a liveness property.

In general, by our experience and knowledge of RMC, we have the impression that RMC is well suited for some domain specific problems. But RMC in general is badly suited for verification of general problems and algorithms, as state space of real world systems are oftently not regular. On basis on our experiments we suggest to continue working with RMC as a method for verification of CA algorithms.

We have also defined the CA to be nondeterministic. However in most literature CA are defined to be deterministic. Often when analysing CA we are not only interested in the forward reachability question, but also backward reachability: "what initial configuration can lead to this configuration". When working with deterministic CA, this is a much more interesting and harder question, as the problem of backwards reachability often is of nondeterministic nature. We have however already considered nondeterminism when doing forward reachability. Our reduction can hence easily be used for backwards reachability analysis, by simply inverting the transducer, and performing the verification as usual.

7.2 Future Work

There are several questions in the thesis that could be interesting to examine in greater details. First of all, in Section 3.4 we gave a short sketch of how we can detect if a non-length preserving transducer can be converted into a equivalent length preserving transducer, and preserve the possibility of verifying liveness questions. It would be interesting to examine this in greater details and carry on some experiments.

We have in this thesis shown that it is possible to perform parametric verification of CA. But we also encounter some problems.

First of all, it would be interesting to examine other ways to encode CA into RMC. We suggest trying a mini-step encoding, where the update of one cell corresponds to one run of the transducer. This mini-step encoding might also be useful for verifying CA with dimensions higher than one.

When verifying FSSP we ran into problems, with a non-regular state space. We have previously in [8], suggested extending RMC to Visibly Pushdown Language (VPL) model checking. The VPL language class is more expressive than regular languages yet it has the same closure properties. In VPL we can e.g. recognise sets of strings described by $a^n b^n$, where $n \in \mathbb{N}_0$. This is useful when we have systems where we want to check if two counters have the same value, or as in the case of FSSP where we need to find the middle of a given string. At that point when we first suggested this extension we did not have any use case to motivate the work. But the work with CA, has shown to be a good use case for VPL model checking. Notice that just by using VPL model checking, it will not solve the problems with the FSSP, we would need to change parts of the encoding for VPL model checking to make it work.

It seems also be interesting to examine how RMC can be used for bounded model checking. By the results that *RMC-finite* is PSPACE-complete we know that the problem is decidable. Now it is just at matter of accelerating the calculation of the result. The bounded model checking can be done in other ways than restricting the initial set. It is imaginable that it is possible to develop an acceleration technique that calculates the effect of applying a transducer n times, instead of an arbitrary number of times.

We believe based on our work with RMC, that RMC is well suited for verification of some special purpose systems. However we do not believe that RMC will ever become suitable for verification of general purpose programs. In spite of this RMC is not an outworn theory and we believe that many interesting appliances and results has yet to be discovered.

Appendix A

Implementation

A.1 Implementation

This section serves as a overview over the prototype implementation of the reduction tool. The prototype tool is divided in several sub systems:

- Java CA text-file parser
- Java RMC framework
- Java CA to RMC reduction tool

In extension to our one tool, we use the ARMC tool, and the FSA Prolog library. In the following we will describe each part of the implementation.

FSA Library FSA [27] is a Prolog library for Finite State Machines (NFA). It implements different types of NFA and transducers, and there most common operations. The library also implements a function for drawing the automata using dot files and functions for reading and writing an internal file format for automatons.

ARMC tool The tool ARMC is a prototype implementation of the acceleration technique proposed by Bouajjani et al. [6] and Habermehl and Vojnar [13].

The implementation is done by done by A. Bouajjani, P. Habermehl, T. Vojnar, and they have kindly lend us there work, for testing out ideas.

We have primly used the [6] part of the tool, as this is one of the fastest methods known in the field of RMC. [13]. The acceleration method used in [6] is described in Section 2.4.

The implementation of the ARMC tool is done using YAP Prolog, and the FSA library for automatons.

Java RMC Framework This is a Java class library for regular model checking. It facilitates the building of transducers and automata, as a Java Objects. The class library can output a transducer and automatons, that can be read directly by the FSA library.

Java CA Framework This framework is a Java class library for Cellular automaton. It contains classes for building cCA and bCA. It also contains implementation of reduction algorithm for cCA and bCA to RMC.

CA parser The CA parser is a small Java parsing for parsing a text-file describing a CA. The parser is used for building a model of the CA using the Java CA Framework.

The parser reads two types of files: A simple version where rules are specified as "a b c -> d", similar to the notation $abc \rightarrow b'$, and a more complex version where the same rules is specified as " b a c d". The complex version also allows the usage of wild cards that expand to all symbols. In general the parser only work for neighbourhood $\{+1, -1\}$ but could easily be extended to larger neighbourhoods. For implementation of the ABP, we have implemented a special method for generating the CA rules.

A.2 Encoding of Rule χ

We have implemented Rule χ as defined in Section 4.2.1. However as it is a bounded encoding, we must consider the border cases. We have simply chosen, to let the border preserve its state by adding the rules:

$\$\ 1\ ?\ \rightarrow 1$

$\$\ 0\ ?\ \rightarrow 0$

$?\ 0\ \$ \rightarrow 0$

$?\ 1\ \$ \rightarrow 1$

where $ is the barricade symbol, and ? is the wild card symbol.

A.3 The Alternating Bit Protocol

This section describes the implementation of the *Alternating Bit Protocol* (ABP). The ABP is a simple network protocol that ensures that messages are delivered to a receiver in the correct order on a lossy or corrupting channel. It does so by altering, or flipping a single sequence bit for each data message. When the sender sends a message with a sequence bit value, the receiver should, when the message is received, send an acknowledgement with a corresponding sequence bit value. Since the channel is lossy, the Sender does not know if the message is going to be lost, so it will resend the message until it receives an acknowledgement with the correct sequence bit value. Then the sender flips the bit and sends a new messages in the same way, but with the new sequence bit value.

Since the sender resends the same message multiple times, it may occur that the sender receives an acknowledgement for one data message more than once. In these cases the sender will simply ignore the acknowledgements for the old message.

A.3.1 Modelling the ABP as a State System

We will model the ABP with a CA and verify it using RMC. For that we will describe the sender states, S_0, S_{a0}, S_1, S_{a1}, the receiver states, R_0, R_{d0}, R_1, R_{d1}, and the messages and acknowledgements, M_0, M_1, A_0, A_1.

First we will have a look at the sender states, S_0, S_{a0}, S_1,S_{a1}:

S_0: In this state, the sender can send one data message with sequence bit 0, M_0; this message can be resend in this state. In this state the sender acts on received acknowledgements for messages send with sequence bit 0, A_0. When such an acknowledgement is received it goes to state S_{a0}. If it receives an acknowledgement with sequence bit 1 A_1, the acknowledgement is ignored and we stay in S_0.

S_{a0}: In this state, where an acknowledgement for a message with sequence bit 0 has been received, the sender is waiting for a new message. When such is arrived it goes to state S_1 and sends a message with sequence bit 1 M_1.

S_1: In this state, the sender can send one data message with sequence bit 1, M_1; this message can be resend in this state. In this state the sender acts on received acknowledgements for messages send with sequence bit 1, A_1. When such an acknowledgement is received it goes to state S_{a1}. If it receives an acknowledgement with sequence bit 1 A_1, the acknowledgement is ignored and we stay in S_1.

S_{a1}: In this state, where an acknowledgement for a message with sequence bit 1 has been received, the sender is waiting for a new message. When such is arrived it goes to state S_0 and sends a message with sequence bit 0 M_0.

Now let us have a look at the receiver R_0, R_{d0}, R_1, R_{d1}:

R_0 In this state the receiver will wait for a data message with sequence bit 0 M_0, to arrive. Then it will go to state R_{d0}, where it delivers the message. If it receives a message with sequence bit 1 it just sends an acknowledgement with that sequence bit A_1.

R_{d0} In this state the receiver sends the message to the next layer and sends and an acknowledgement with sequence bit 0 A_0 and goes to state R_1.

R_1 In this state the receiver will wait for a data message with sequence bit 1 M_1, to arrive. Then it will go to state R_{d1}, where it delivers the message. If it receives a message with sequence bit 0 it just sends an acknowledgement with that sequence bit A_0.

R_{d1} In this state the receiver sends the message to the next layer and sends and an acknowledgement with sequence bit 1 A_1 and goes to state R_0.

The transition system for the sender and the receiver is illustrated in Figure A.1. Here we can see the channel in between the sender and the receiver.

Now the thing we would like to verify is that the sender can not be in state S_{a0} and the receiver in state R_{d1} since that means that the receiver have received a message with sequence bit 0 and while the sender has send a message with sequence 1.

A.3.2 Modelling ABP and the Channel as bCA

To model the channel in a CA, we present the channel as a notion of a two way tape. The messages will move right in the "top-tape" and the acknowledgements will move left in the "bottom-tape". We simulate the two way channel by combining the data message states and the acknowledgement states and introducing an empty-state '$-$'. E.g. a data message with sequence bit 0 on the tape will be the states: $\binom{M_0}{-}$, $\binom{M_0}{A_0}$ and $\binom{M_0}{A_1}$. The rules to model the channel and the sender and receiver id presented in Table A.1. Here we use wild card symbols again and the symbols X and Y to denote a wild card that is going to be fixed within one rule i.e. the rule $\binom{X}{?}\binom{?}{?}\binom{?}{Y} \rightarrow \binom{X}{Y}$ means that we do not care what state all the cells are in only that X moves right and Y moves left.

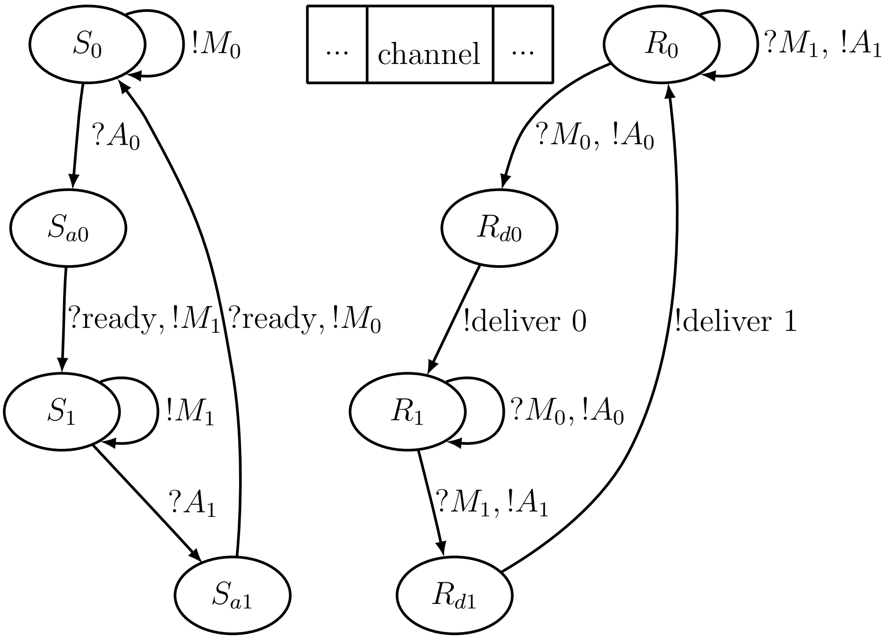


Figure A.1: This diagram illustrates the states of the sender and the receiver. Here transitions that start with a '!' is output and '?' means output. The channel can "handshake" with M_0, M_1, A_0 and A_1 and deliver and ready transitions "handshakes" with other layers.

Now the question we would like to verify is:
From configurations $\big(\$, S_0, \binom{?}{?}, \ldots, \binom{?}{?}, R_0, \$\big)$, can vi avoid configurations $\big(\$, S_{a0}, \binom{?}{?}, \ldots, \binom{?}{?}, R_{d1}, \$\big)$?

#	Rule
1	$S_0\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{-}{Y}$
2	$S_0\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{M_0}{Y}$
3	$\$S_0\binom{?}{\_} \longrightarrow S_0$
4	$\$S_0\binom{?}{A_0} \longrightarrow S_{a0}$
5	$\$S_0\binom{?}{A_1} \longrightarrow S_0$
6	$S_{a0}\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{M_1}{Y}$
7	$\$S_{a0}\binom{?}{?} \longrightarrow S_1$
8	$S_1\binom{?}{?}\binom{-}{Y} \longrightarrow \binom{-}{Y}$
9	$S_1\binom{?}{?}\binom{-}{Y} \longrightarrow \binom{M_1}{Y}$
10	$\$S_1\binom{?}{\_} \longrightarrow S_1$
11	$\$S_1\binom{?}{A_1} \longrightarrow S_{a1}$
12	$\$S_1\binom{?}{A_0} \longrightarrow S_1$
13	$S_{a1}\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{M_0}{Y}$
14	$\$S_{a1}\binom{?}{?} \longrightarrow S_0$
15	$\binom{X}{?}\binom{M_1}{?}R_0 \longrightarrow \binom{X}{A_1}$
16	$\binom{X}{?}\binom{?}{?}R_0 \longrightarrow \binom{X}{-}$
17	$\binom{-}{?}R_0\$ \longrightarrow R_0$
18	$\binom{M_0}{?}R_0\$ \longrightarrow R_{d0}$
19	$\binom{M_1}{?}R_0\$ \longrightarrow R_0$
20	$\binom{X}{?}\binom{?}{?}R_{d0} \longrightarrow \binom{X}{A_0}$
21	$\binom{?}{?}R_{d0}\$ \longrightarrow R_1$
22	$\binom{X}{?}M_0?R_1 \longrightarrow \binom{X}{A_0}$
23	$\binom{X}{?}\binom{?}{?}R_1 \longrightarrow \binom{X}{-}$
24	$\binom{-}{?}R_1\$ \longrightarrow R_1$
25	$\binom{M_0}{?}R_1\$ \longrightarrow R_1$
26	$\binom{M_1}{?}R_1\$ \longrightarrow R_{d1}$
27	$\binom{X}{?}\binom{?}{?}R_{d1} \longrightarrow \binom{X}{A_1}$
28	$\binom{?}{?}R_{d1}\$ \longrightarrow R_0$
29	$\binom{X}{?}\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{X}{Y}$
30	$\binom{X}{?}\binom{?}{?}\binom{?}{Y} \longrightarrow \binom{-}{-}$

Table A.1: CA-rules for the alternating bit protocol as a CA.

Bibliography

[1] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. *Lecture Notes in Computer Science*, 2404:555, 2002.

[2] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. 2002.

[3] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for LTL(MSO). In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2004.

[4] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[5] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. *Lecture Notes in Computer Science*, 2725:223–235, 2003.

[6] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. *Lecture Notes in Computer Science*, 2004(3114):372–386, 2004.

[7] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[8] Joakim Byg and Kenneth Yrke Jørgensen. Regular model checking. Department of Computer Science Aalborg University: Project library, 2008.

[9] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. In *Foundations of Software Technology and Theoretical Computer Science*, pages 326–337, 1993.

[10] Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. *Lecture Notes in Computer Science*, 2102:286, 2001.

[11] Massimo D'Antonio and Giorgio Delzanno. Sat-based analysis of cellular automata. In *ACRI*, pages 745–754, 2004.

[12] E. Goto. A minimal time solution of the firing squad problem. In *Dittoed course notes for Applied Mathematics*, volume 298, pages 52–59, 2062.

[13] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages, 2004.

[14] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2):93–112, 2001.

[15] F klub Aalborg University. F-klub.
`http://fklub.cs.aau.dk`.

[16] Jacques Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theor. Comput. Sci.*, 50:183–238, 1987.

[17] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, Berlin, 1980. (LA has).

[18] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[19] David Moews. The finite state firing squad.
`http://djm.cc/fsquad/firing.html`.

[20] Kai Nagel. Particle hopping models and traffic flow theory. *Phys. Rev. E*, 53(5):4655–4672, May 1996.

[21] John Von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[22] Marcus Nilsson. Regular model checking.
`http://regularmodelchecking.com`, 2004.

[23] Princeton University SAT Research Group. zchaff.
`http://www.princeton.edu/~chaff/zchaff.html`.

[24] Michael Sipser. *Introduction to the Theory of Computation, Second Edition International Edition*. Thomson Course Technology, 25 Thomson Place, Boston, Massachusetts, 02210, USA, 2006.

[25] Tayssir Touili. Regular model checking using widening techniques. *Electr. Notes Theor. Comput. Sci*, 50(4), 2001.

[26] Hiroshi Umeo1 and Takashi Yanagihara. A smallest five-state solution to the firing squad synchronization problem. In *Lecture Notes in Computer Science*, pages 291–302, 2007.

[27] Gertjan van Noord. Fsa6.2xx: Finite state automata utilities.
`http://www.let.rug.nl/~vannoord/Fsa/`, 2008.

[28] Wikipedia. Rule 110.
`http://en.wikipedia.org/wiki/Rule_110`.

[29] Wikipedia.com. Conways game of life.
`http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`, 2008.

[30] Stephen Wolfram. Cryptography with cellular automata. In *Advances in Cryptology*, page 429, 1985.

[31] Wolper and Boigelot. Verifying systems with infinite but regular state spaces. In *CAV: International Conference on Computer Aided Verification*, 1998.

[32] YAHODA. Yahoda - verification tools database.
`http://anna.fi.muni.cz/yahoda/`, 2008.