Model Checking Cellular Automata

Joseph A. Gershenson

SCS Undergraduate Thesis

Advisor: Klaus Sutner

May 1, 2009

Contents

Acknowledgments

This thesis could not have been completed without the help of many others. First, I would like to thank my friends and family for their continued support and encouragement through the writing process, as well as throughout my educational career.

Second, I am extremely grateful to Klaus Sutner for the opportunity to work with him, both over the past two years and next year in the Fifth Year Scholars program. He continues to be a great teacher, an excellent advisor, and a wonderful guide to the world of theoretical computer science.

Finally, my heartfelt thanks also go to Mark Stehlik for the tremendous amount of work he has done to improve my experience at Carnegie Mellon over the past four years. A tireless advocate of undergraduate students, he has been opening doors for my class since before our arrival on campus.

Abstract

Simple aspects of the evolution of one-dimensional cellular automata can be captured by the first-order theory of phase-space, which uses one-step evolution as its main predicate. Formulas in this logic can thus be used to express statements such as "there exists a 3-cycle" or properties of the global map such as surjectivity. Since this theory has been shown to be decidable by using two-way infinite Büchi automata, it is possible to evaluate these formulas by manipulating the Büchi automata. We implement such a system and report on the results as well as the tractability of larger problems.

Chapter 1

Introduction

Cellular automata are valuable systems for modeling computational processes because of their simplicity and ability to represent the behavior of complex systems. However, the same power that makes cellular automata useful also makes them difficult to analyze. Model checking, or the use of methods for formally specifying and verifying the behavior of advanced systems, is a powerful tool for computer scientists today. We describe the implementation of a model-checking procedure for cellular automata.

The theory inspiring this paper is presented by Sutner in [19] as a constructive proof that model-checking cellular automata is decidable. By model-checking cellular automata, we refer to the analysis of properties of the global map of the cellular automaton using formal methods. Sutner's theory specifies properties of cellular automata as formulae in the first-order theory of phase-space, and provides a method for evaluating these properties. Because exponential and super-exponential constructions are required when evaluating these properties, it was unclear whether this method would be feasible for real problems. Our implementation answers this question in the affirmative, but also identifies current boundaries to our capabilities.

The natural predicate in the theory of phase-space is the global map of the cellular automaton. To check predicates in this theory, we construct a Büchi automaton which decides whether two bi-infinite words satisfy a particular relation. The characters in the alphabet of such an automaton consist of one character from each of the two component words. For example, the equality relation $=(A, B)$ can be checked by an trivial machine which accepts only words in which both of the words A and B have the same character at the each index. The relation $\rightarrow(A, B)$ corresponding to the global map requires a more complex automaton, the construction of which is detailed below. Construction of more complicated formulas in the theory is done inductively by performing appropriate operations on the automata representing the appropriate sub-formulae.

Our implementation of the model-checking procedure uses Mathematica for high-level specification and interface purposes. The low-level construction of automata is done in Java for performance purposes and results are returned to Mathematica via JLink. To exemplify the requirements for an efficient implementation, we note that complementation of a formula requires determinizing the corresponding Büchi automaton. Running Safra's determinization algorithm on an automaton of n states can generate an automaton of $2^{O(n \log n)}$ states [15]. Since complementation is required for most formulas, this means that tractability is a serious concern for model-checking cellular automata. Accordingly, we will also present several improvements to the determinization algorithms which help to reduce the size of the resulting machines.

Chapter 2

Automata Theory

The study of automata on infinite words began in the 1960s, and was originally motivated by the desire to solve abstract problems in second-order logic. In recent years, the focus has shifted to the use of these automata in model-checking concurrent systems. We refer the reader to [20] for a canonical source, and also to [12] for an accessible and updated reference. In this chapter, we present definitions of the automata involved in this thesis, and will provide a brief background as to their construction and use. There is some variety regarding terminology in the literature; we will seek to keep our definitions as consistent as possible by using these references as standards.

2.1 Definitions and Background

John von Neumann was the first to describe cellular automata, while searching for a formalization of self-reproducing structures. Since their inception, cellular automata have been used for a variety of modeling problems; an excellent history of the field is presented in [16]. Variant definitions and extensions of the theory of cellular automata have been developed, so we clarify our definition of a cellular automaton below.

The ω-automata are extensions of the concept of a finite automaton to one-way infinite inputs. Their use is one of the simplest natural ways of describing and recognizing infinite words. The apparent similarity of a "cellular automaton" to a "finite automaton" or "Büchi automaton" is an unfortunate namespace collision; the cellular automaton is quite a distinct concept from the classical automata we are about to discuss. We will endeavor to be as disambiguating as possible, but the word *automaton* alone never refers to a cellular automaton.

2.1.1 Cellular Automata

Informally, a cellular automaton represents a set of cells and a rule which dictates their evolution over time. In a simple example, we arrange the cells in a one-dimensional line. Time occurs in discrete steps, and the contents of each cell are updated at each step depending on the concept of their neighbors. To view the evolution of these cells over time, we arrange the one-dimensional rows above and below each other. An example is given below in Figure 2.1, where we show the evolution of a configuration beginning with a single black cell. Under the rule "assume the value of your right neighbor," the location of the black cell shifts to the left. This rule, or *local map*, corresponds to Wolfram's Rule 2 for elementary cellular automata.

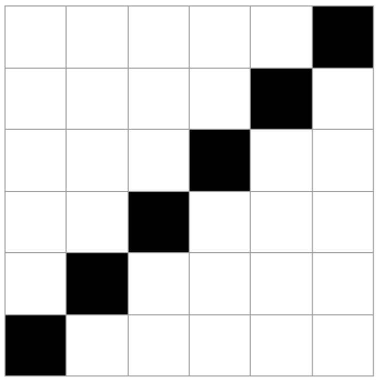

Figure 2.1: The evolution of a simple cellular automaton.

We are now prepared to formally define this concept:

Definition A *cellular automaton* is a local map $\rho : \Sigma^{2r+1} \rightarrow \Sigma$ where $r \geq 0$ is the *radius* of the automaton and Σ the *alphabet*. The radius of a cellular automaton is the maximum distance at which a neighboring cell may influence a cell's content at the next time step. The alphabet of a cellular automaton is the set of possible contents for a cell.

Since the local map of a cellular automaton is a function from Σ^{2r+1} to Σ, all cellular automata of a fixed alphabet and radius are enumerable. Wolfram defines a simple enumeration for all the cellular automata of radius 1 and alphabet $\{0, 1\}$ in [23]; these automata are often referred to as the *elementary cellular automata*, and form a natural starting point for examining properties of cellular automata in general.

Certain extensions to our definition of cellular automata are immediately apparent. First, it is worth noting that cellular automata may be defined in an arbitrary number of dimensions. The most famous cellular automaton, Conway's Game of Life, is two-dimensional. In this thesis, we will confine ourselves to the discussion of one-dimensional cellular automata out of necessity, since cellular automata in higher dimensions are not amenable to the property-checking algorithms that we wish to demonstrate. Automata may also have a larger alphabet or a larger radius than in our example, so that the cells themselves have more than two possible values for their contents or the local map references the value of more cells. We will discuss how our methods apply to working with automata of various alphabets and radii, but in general the assumption that we are dealing with the elementary cellular automata is a useful and simplifying one.

We have thus far neglected the significant issue of the *boundary conditions* for the environment of the cellular automaton. The boundary conditions may be finite, so that the total number of cells is some finite number. When implementing such boundary conditions, it is common practice for the terminal cells in either direction to be treated as adjacent, so that the line of cells forms a loop and a sufficient number of arguments can be passed to the local map at each index. Another common practice is to assume that all cells past the boundary have the value of zero, so that the values of the local map will be constrained at the terminal cells.

Another natural boundary condition is a one-way or two-way infinite line of blank cells. The initial configuration for an automaton with these boundary conditions is thus a finite number of non-empty cells surrounded by infinitely many empty cells in one or both directions. The configuration could also be periodic, so that the infinitely recurrent pattern is nonempty in one direction or both direction. In order to describe these configurations, we need to develop tools to recognize infinite and bi-infinite words and languages: this will be one motivation for our introduction of ω- and ζ-automata.

Definition A *configuration* is a function $C \to \Sigma$ relating the cells of the automaton to characters from the alphabet. It specifies the contents of each cell present in the automaton at a given time step. A configuration is therefore representable as a finite word when the corresponding automaton has finite boundary conditions, or as an infinite word when the corresponding automaton has infinite boundary conditions.

We are naturally interested in the evolution of configurations of cellular automata over time. The formal discussion of properties associated with this evolution is aided by the concepts of the *global map* and *phase-space*:

Definition The *global map* $G_\rho : \Sigma^n \to \Sigma^n$ of a cellular automaton with finite boundary conditions is the extension of the local map ρ to the entire configuration. Extending the local map in an automaton with an infinite boundary condition allows us to define a global map $G_\rho : \Sigma^{\mathbb{Z}} \to \Sigma^{\mathbb{Z}}$ (or $G_\rho : \Sigma^{\mathbb{N}} \to \Sigma^{\mathbb{N}}$, as appropriate). For notational convenience and clarity, we will write $x \xrightarrow{\rho} y$ for $G_\rho(x) = y$.

Definition The *phase-space* of a cellular automaton ρ is the functional digraph of the global map G_ρ. In other words, the phase-space is a directed graph (V, E) where every vertex $v \in V$ corresponds to a unique configuration of the automaton, and each edge $(u, v) \in E$ is present if and only if $G_\rho(u) = v$.

It is easy to see that interesting properties of cellular automata may be characterized as statements about the phase-space. For example, suppose that we are interested in determining whether the evolution of cellular automaton ρ results in a 3-cycle: a set $\{x, y, z\}$ of configurations such that $x \xrightarrow{\rho} y \xrightarrow{\rho} z \xrightarrow{\rho} x$. This property naturally translates into the assertion "the phase-space contains a 3-cycle."

If a cellular automaton has finite boundary conditions, the phase-space is a finite graph which can be completely enumerated, so such a property can be checked by a graph traversal algorithm such as depth-first search. Cellular automata with infinite boundary conditions, however, have an uncountable number of configurations and thus an uncountably infinite graph for the phase-space. Validating assertions about the phase-space of cellular automata with infinite boundary conditions therefore requires a different strategy. We will build up this strategy in the following sections; its implementation is the primary aim of this thesis. First, though, we focus on a vital component: the automata which are used to recognize infinite words.

2.1.2 ω-Automata

To properly define ω-automata, we first review the concept of a finite automaton. Finite automata, used to recognize words over finite strings, should be familiar from many applications in computer science.

Definition An *automaton* is a tuple (Q, Σ, δ) where Q is the set of states, Σ is the alphabet, and $\delta \subset Q \times \Sigma \times Q$ is the transition relation. This definition represents the basic transition system which is common to all automata. To fully define a finite automaton, we require some notion of acceptance.

Definition A *finite automaton* is an automaton defined with a set of initial states $I \subset Q$ and a set of final states $F \subset Q$, to form a 5-tuple $(Q, \Sigma, \delta, I, F)$.

An automaton is used to accept or reject words over its alphabet Σ, which we can formalize using the concept of a *run*.

Definition A *run* of an automaton (Q, Σ, δ) over a finite word $w \in \Sigma^n$, where $w = w_0 w_1 ... w_{n-1}$, is a sequence $q_0, q_1, ..., q_n$ such that all transitions $(q_i, w_i, q_{i+1}) \in \delta$ for $0 \leq i < n$. For a finite automaton, a run is *accepting* if and only if $q_0 \in I$ and $q_n \in F$.

Armed with the concept of runs, we formally define what it means to recognize a word. A word w is *recognized* by an automaton A if there is at least one accepting run on A for w. The *language* $L(A)$ of an automaton A is the set of words recognized by A. A set of finite words is *recognizable* if and only if it is the language of some finite automaton. A finite automaton A is *empty* if $L(A) = \emptyset$ and *universal* if $L(A) = \Sigma^*$.

Figure 2.2 shows an example of a simple finite automaton, which is formally defined by $(\{1,2\}, \{a,b\}, \{(1,a,1), (1,b,1), (1,b,2)\}, \{1\}, \{2\})$. This automaton recognizes the language $(a+b)^*b$, or the set of all strings over $\{a,b\}$ which have b as a terminal character. This machine also demonstrates two properties that automata may have. First, it is incomplete; not all possible transitions are defined from every state. This is compatible with our definition of a finite automaton above; we say that a run *fails* and may not be accepting if one of the necessary transitions is undefined. Second, and more importantly, it is nondeterministic: not all transitions are unambiguous.

Definition An automaton $(Q, \Sigma, \delta, I, F)$ is *nondeterministic* if there is at least one state $q \in Q$ and one character $\sigma \in \Sigma$ for which $(q, \sigma, q') \in \delta$ and $(q, \sigma, q'') \in \delta$ with $q' \neq q''$. The automaton is also nondeterministic if $|I| > 1$.

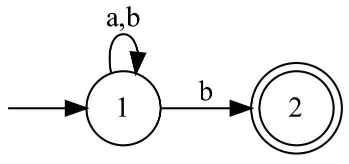

Figure 2.2: A simple finite automaton with alphabet $\{a, b\}$.

All automata on one-way infinite inputs are referred to as ω-automata. They scan words over Σ^ω much as finite automata scan words over Σ^n. In addition, ω-automata differ from finite automata in their acceptance conditions; we will see this best by example.

Definition A Büchi automaton is a tuple $(Q, \Sigma, \delta, I, F)$ where (Q, Σ, δ) is an automaton, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states.

A Büchi automaton is the simplest extension of the theory of finite automata to one-way infinite strings; its definition is virtually identical to that of a finite automaton. Before discussing the languages recognized by Büchi automaton, we must redefine a run for the one-way infinite case.

Definition A *run* of an automaton (Q, Σ, δ) on an infinite word $w \in \Sigma^\omega$, where $w = w_0 w_1 ...$, is a sequence $q_0, q_1, ..., q_n, ...$ such that all transitions $(q_i, w_i, q_{i+1}) \in \delta$ for $i \in \mathbb{N}$. A state q is *infinitely recurrent* in this run if, for all $i \in \mathbb{N}$, there exists some $j > i$ such that $q = q_j$.

For a Büchi automaton, a run is accepting if any state in F is infinitely recurrent. A word is recognized by a given ω-automaton if there is at least one accepting run, and, as in the finite case, the language of a ω-automaton remains the set of words recognized by that automaton. Büchi automata are also a natural way to define the recognizability of sets of infinite words; a language $L \subset \Sigma^\omega$ is recognizable if and only if there is some Büchi automaton A such that L is the language of A. An ω-automaton A is *empty* if $L(A) = \emptyset$ and *universal* if $L(A) = \Sigma^\omega$.

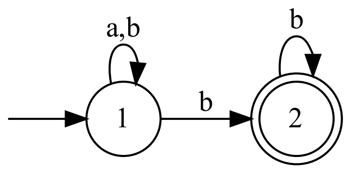

Figure 2.3: A simple Büchi automaton.

An example of a Büchi automaton is given in Figure 2.3. This machine recognizes all infinite words over $\{a, b\}$ which contain only finitely many a's. It also exemplifies an important property of Büchi automata: a word is not necessarily recognized if an final state is reached at every time step by a different run. It is necessary for a single run to exist which reaches a final state infinitely often. This distinction is important when the automaton in Figure 2.3 is run on the word $(ab)^\omega$; for all $i \in \mathbb{N}$ there is a run of the automaton which reaches state 2 at time $2i$. However, these runs all fail at time $2i + 1$.

For finite automata, we can always construct a deterministic automaton which recognizes the same language as a nondeterministic automaton by using a power set construction. However, the same is not true of Büchi automata: there exist recognizable languages L such that L is not the language of any deterministic Büchi automaton. A formal proof, provided in [12], relies on the notion of prefixes. While recommended to the interested reader, it is beyond the scope of this thesis. The critical result for now is that a new type of ω-automaton must be defined if we are to have a deterministic ω-automaton equivalent in computational power to a Büchi automaton.

Definition A *Rabin automaton* is a tuple $(Q, \Sigma, \delta, i, R)$ where (Q, Σ, δ) is an automaton, i is the the initial state, and $R = \{(E_j, F_j)\}$ where $E_j, F_j \subset Q$ represents the acceptance condition, a set of *Rabin pairs*.

A Rabin automaton is deterministic by definition, so δ defines at most one transition (q, σ, q') for each (q, σ), and the initial state $i \in Q$ is unique. The acceptance condition $R = \{(E_j, F_j)\}$ is a set of pairs of sets of states. A run

$r = q_0, q_1, \ldots$ of a one-way infinite word is accepting if there exists some index j such that r reaches F_j infinitely often and reaches E_j only finitely often. The equivalence of Büchi automata and Rabin automata will be discussed in Section 2.2.1. As an example, a Rabin automaton accepting the same language as the Büchi automaton we saw earlier, that of only finitely many a's, is presented in Figure 2.4.

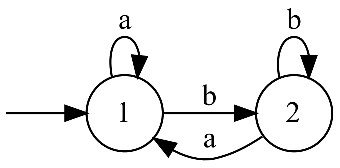

Figure 2.4: A simple Rabin automaton. The acceptance condition R is $\{(\{1\}, \{2\})\}$, so an accepting run is one which reaches state 1 finitely often and state 2 infinitely often.

2.1.3 ζ-Automata

While there is some room for opinion about the optimal representation of ω-automata, there is a broad spectrum of definitions of ζ-automata. We will follow [19] and [8] in our definition, which strives to be as simple and conventional as possible. The resulting ζ-automaton is familiar after our review of finite and ω-automata: a ζ-Büchi automaton is a generalization of a Büchi automaton to accept two-way infinite words.

Definition A *ζ-Büchi automaton*, or *ζ-automaton*, is a tuple $(Q, \Sigma, \delta, I, F)$ where (Q, Σ, δ) is an automaton, $I \subset Q$ is the set of *initial* states, and $F \subset Q$ is the set of *final* states. A *run* of an automaton (Q, Σ, δ) on a bi-infinite word $w \in \Sigma^{\mathbb{Z}}$, where $w = \ldots w_{-1} w_0 w_1 \ldots$, is a bi-infinite sequence $\ldots, q_{-n}, \ldots, q_{-1}, q_0, q_1, \ldots, q_n, \ldots$ such that all transitions $(q_i, w_i, q_{i+1}) \in \delta$ for $i \in \mathbb{Z}$. A state q is *infinitely recurrent* in this run if, for all $i \in \mathbb{Z}$, there exists some $j > i$ such that $q = q_j$, and q is *infinitely precurrent* in this run if, for all $i \in \mathbb{Z}$, there exists some $j < i$ such that $q = q_j$.

Let r be a run of ζ-automaton $Z = (Q, \Sigma, \delta, I, F)$ on some bi-infinite word w, with r_I and r_F the sets of infinitely precurrent and recurrent states, respectively. Run r is accepting if $r_I \cap I \neq \emptyset$ and $r_F \cap F \neq \emptyset$. A bi-infinite word w is recognized by a ζ-automaton A if there is an accepting run of A on w, and the set of words recognized by A is the language $L(A)$ of A. A ζ-automaton A is *empty* if $L(A) = \emptyset$ and *universal* if $L(A) = \Sigma^{\mathbb{Z}}$.

This definition of a ζ-automaton allows us to specify precurrent and recurrent patterns in the bi-infinite word, and thus gives us a basis for describing languages of bi-infinite words. A key distinction from the one-way infinite case is that bi-infinite words have no natural coordinate system. While a word over $\Sigma^{\mathbb{N}}$ has a natural first character, and thus each character can be indexed, there is no natural reference point for words over $\Sigma^{\mathbb{Z}}$. Bi-infinite words are often described as having the property of *shift-equivalence* for this reason, since a bi-infinite word where every character is shifted one place to the right has not changed at all.

2.2 Algorithms and Operations

In order to obtain useful results with automata on infinite words, a few common operations are required. Many of these are relatively, such as union and intersection, and revolve around performing the corresponding operation on the state set and transition relations of the automata. The operations of determinization and complementation, in particular, require a bit more consideration, primarily because of their computationally significant cost.

2.2.1 Determinization of ω-Automata

Determinization refers to the generation of an equivalent deterministic automaton given a nondeterministic one. For finite automata, the classical result mentioned earlier was given by Rabin and Scott in [14]: for every nondeterministic finite automaton, we can construct a deterministic finite automaton which accepts precisely the same language. We review the Rabin-Scott construction briefly for the sake of demonstrating its inapplicability to automata on infinite words.

The Rabin-Scott construction uses a power set construction, where each state in the deterministic automaton represents a set of states in the nondeterministic automaton. If $\{q_0, q_1, ..., q_n\}$ is the set of all runs of the nondeterministic automaton on a given word, then $\{d_0, d_1, ..., d_n\}$ is the run on the deterministic automaton, where $d_i = \bigcup q_i$. Thus, given a nondeterministic automaton $(Q, \Sigma, \delta, I, F)$, the nondeterministic automaton $(2^Q, \Sigma, \delta', I', F')$ accepts the same language, where

$$\begin{aligned}
\delta' = &\quad \{(Q_1, \sigma, Q_2) \mid Q_2 = \bigcup_{q \in Q_1} \{q' \mid (q, \sigma, q') \in \delta)\}\} \\
I' = &\quad \{I\} \\
F' = &\quad \{q \mid q \cap F \neq \emptyset\}
\end{aligned}$$

The power set construction is simple and effective, but also exponential in the worst case. Determinizing a finite automaton of n states therefore requires $O(2^n)$ time and produces an automaton of $O(2^n)$ states in the worst case.

As hinted previously, the Rabin-Scott power set construction is ineffective for determinizing automata on infinite words. We can informally explain this by returning to the example automaton in Figure 2.3. A power set construction on this automaton would record the states that any run could be in at a given time. However, the automaton which resulted from a power set construction would be incapable of distinguishing which run reached a final state at a given time. When the automaton scans the word $(ab)^\omega$, for example, there is a run which reaches state 2 after every other character. This run always fails immediately afterwards, however, on the following a. The power set automaton thus alternates between states labeled by $\{1\}$ and $\{1, 2\}$, and since $\{1, 2\}$ is a final state, the run would be accepting and the machine would incorrectly recognize the word.

From the failure of this attempt to determinize a Büchi automaton, we can see that it is necessary to somehow record which run is reaching a final state at a given time. Only by doing this can we ensure that a single run reaches a final state infinitely often. As we briefly referenced above, we know that a deterministic Büchi automaton cannot recognize every recognizable language, so we look for a way to convert a Büchi automaton into a Rabin automaton. The equivalence of Büchi and Rabin machines for recognizing infinite words is given by R. McNaughton in [9]:

Theorem 2.1. *Any recognizable subset of Σ^ω can be recognized by a Rabin automaton.*

A constructive proof of McNaughton's Theorem, given by S. Safra, describes a method for computing an equivalent Rabin automaton given a

Büchi automaton. Conceptually, the algorithm memorizes occurrences of final states, and records the points at which a given run returns to a final state in order to ensure that a singular run in the Büchi automaton reaches a final state infinitely often. We will omit a formal proof of correctness due to length; the interested reader is referred to [15, 12]. Safra's determinization algorithm is presented here for reference and because improvements to this algorithm feature prominently in our work.

Safra's Construction

Given a Büchi automaton $(Q, \Sigma, \delta, I, F)$, we build a Rabin automaton $R = (T, \Sigma, \delta, I, F)$ where the elements of T correspond to labeled trees. At a high level, we will explore the state set of R by repeatedly constructing the resulting tree after a transition in the automaton. Each node v in a tree has a unique name drawn from $[2n]$ (where $|Q| = n$), and is labeled with a nonempty subset of Q, denoted by $L(v)$. Nodes may be *marked* or *unmarked*, and we hold as an invariant that the union of the labels of the children of v is a strict subset of the label of v. The initial state of R is given by a simple tree. If $I \cap F = \emptyset$, the initial tree is one unmarked node labeled with I. If $I \subset F$, the tree is one marked node labeled I. Otherwise, the tree consists of an unmarked node labeled I with a marked child labeled $I \cap F$.

We calculate the state set of R by performing transitions on the states. On character α, we transform tree T as follows:

1. We perform the transition by α on the labels of each node, and erase all marks.

2. For each node v, we create a new rightmost child of v with label $L(v) \cap F$. We mark the new node and assign it a name. In the standard version of Safra's construction, we use the smallest available integer.

3. For all nodes v and v', where v is a left sibling of v', remove all states in $L(v)$ from $L(v')$.

4. Remove all nodes with an empty label.

5. If the union of the labels of the children of v is equal to the label of v, mark v and remove all children of v.

The resulting tree represents a new state of R. We continue performing the transitions until the graph is complete, and we have defined a transition from every state on every character in Σ. The total number of trees is bounded by $2^{O(n \log n)}$, where $n = |Q|$. The algorithm is guaranteed to terminate in $2^{O(n \log n)}$ time . The set F of final states is defined as $\{(E_i, F_i) | i \in 2n\}$, where the state corresponding to tree T is in E_i if i is not the name of any node present in T, and the state corresponding to tree T is in F_i if i is the name of a marked node in T.

From a Büchi automaton with n states, this determinization algorithm allows us to construct an equivalent Rabin automaton with $2^{O(n \log n)}$ states and $O(n)$ pairs of sets of states in the acceptance condition. While far from attractive, this is good enough to render determinization practical in some applications; remember that determinization of an automaton over finite words results in a machine of $O(2^n)$ states. We will review the tractability of Safra's construction in greater detail when discussing our implementation in Chapter 4.

2.2.2 Complementation of ω-Automata

When performing operations on ω-automata and the languages which they recognize, we will frequently be interested in complementing an ω-automaton. Given an ω-automaton A, the goal of a complementation algorithm is to construct an automaton $\bar{A}$ which recognizes $\Sigma^\omega \setminus L(A) = \bar{L}(A)$. A significant amount of work has been done in this area over the last 40 years; Vardi provides a detailed survey in [22].

The best lower bound on the blowup of a Büchi automaton during complementation, established by Yan in [24], demonstrates that in the worst case an automaton of n states requires at least $O((0.76n)^n)$ states in an automaton representing the complement. We will omit presenting the majority of results with respect to the complementation of Büchi automata; the interested reader is referred to [22] and [17]. Two results in particular are worth mentioning here: an algorithm for complementation presented in [12] which makes use of Safra's determinization algorithm, and a direct algorithm for complementation recently presented in [17] which approaches the known lower bound.

Rabin automata, because of their deterministic quality and the nature of their acceptance condition, are naturally more amenable to complementation than Büchi automata. Therefore, the first algorithm we use for complementation begins by converting the Büchi automaton into an equivalent deterministic Rabin automaton via Safra's construction. The construction of a Büchi automaton which recognizes the complement language uses a *cut-point* construction to keep track of the states that have been visited. We will avoid detailing it further here, since we make no improvements to this construction and the complete algorithm and proof of correctness is provided in [12]. For the purposes of this thesis, the important elements are that the algorithm utilizes Safra's construction, and generates an automaton of $2^{O(n \log n)}$ states given an input automaton of n states.

The complementation method proposed by Schewe in [17], by contrast, does not require the determinization of the automaton. It generates a complement automaton of $O(n^2(0.76n)^n)$ states from an input automaton of n states. This approaches the lower bound on complementation of Büchi automaton. However, several factors remain a barrier to the use of this method for our purposes; these are discussed further in Sections 5.1.2 and 6.1.

2.2.3 Complementation of ζ-Automata

A proof that the family of languages recognized by ζ-automata is closed under complementation was discussed in [11], but a constructive algorithm for complementing such a language was not shown until the work of Culik and Yu in [8]. The algorithm relies on the fact that the language recognized by every ζ-automaton is what the authors describe as $\omega\omega$-regular. A $\omega\omega$-regular set is a set which can be written as a finite union of languages of the form ${}^{\omega}ABC^{\omega}$, where A,B,C are all the languages of finite automata. Such an $\omega\omega$-regular set is closed under complementation; the proof, found in [8], requires taking the union of a finite but exponential number of ω-automata.

Intuitively, the complementation algorithm uses ω-automata to represent the languages ${}^{\omega}A$ and C^{ω}. Suppose that we have a ζ-automaton Z recognizing ${}^{\omega}ABC^{\omega}$, and some bi-infinite word $w = {}^{\omega}abc^{\omega}$ which is not in the language of Z. Then, it stands to reason that at least one of $(a \notin A)$, $(b \notin B)$, $(c \notin C)$ is true. Since we can check $(a \notin A)$ by complementing an ω-automaton that recognize ${}^{\omega}A$, we can check to see if at least one of these properties is true for a given bi-infinite word w.

A sticking point here is that a bi-infinite word w may be broken up into many possible factorizations of the form ${}^{\omega}abc^{\omega}$. A ζ-automaton recognizes a word if any run on that word is accepting, so for every possible factorization of w into ${}^{\omega}abc^{\omega}$, at least one of $(a \notin A)$, $(b \notin B)$, $(c \notin C)$ must hold.

The exponential number of ω-automata required comes first from the necessity of checking A and B independently, and second from the fact that a ζ-automaton may recognize words from the union of a finite number of sets ${}^{\omega}ABC^{\omega}$. It is necessary to split these languages into pairwise-disjoint forms before attempting to construct or complement the appropriate ω-automata. The running time of complementation of ζ-automata therefore adds another exponential factor to the running time of complementation of ω-automata.

Chapter 3

Model-Checking for Cellular Automata

Our goal has been to answer questions about the behavior of cellular automata, and we employ the strategy of model-checking to that end. Model-checking, a method of formally verifying assertions about the behavior of systems, is described by Clarke et al. in [4] as consisting of three phases: *modeling*, *specification*, and *verification*. The application of this process to cellular automata allows us to answer our questions by proving properties of phase-space of the automata.

The inspiration for this thesis is primarily due to the following theorem, due to Sutner:

Theorem 3.1. *model-checking for one-dimensional cellular automata is decidable.*

This chapter presents an overview of Sutner's constructive proof of this theorem, using Clarke's phases of model-checking as a framework. The interested reader is also referred to the original exposition of the proof in [19]. Note that in the figures in this chapter, we will often assume the alphabet Σ of a cellular automaton is restricted to $\{0, 1\}$ for simplicity.

3.1 Modeling

In modeling, a design is converted into a formalism which is accepted by a model-checking tool. When applying model-checking to a cellular automaton, this consists of constructing a transition automaton to represent one step in the evolution of the configuration.

The type of automaton used to model the evolution of a cellular automaton is dependent on the boundary conditions in use. A cellular automaton with finite boundary conditions can be checked using a finite automaton. For one-way infinite boundary conditions we use ω-automata, and for two-way infinite boundary conditions we use ζ-automata. We will explain the use of ζ-automata; modeling transitions with finite or one-way infinite boundary conditions is in fact slightly more difficult, even though verification becomes vastly easier. The automata we construct to model the basic transition scan words over Σ^2, where Σ is the alphabet of the cellular automaton ρ that we are modeling.

Formally, given the cellular automaton ρ with alphabet Σ and radius r, we define the transition automaton $T_\rho = (Q, \Sigma^2, \delta, I, F)$. We let $Q = \Sigma^{2r} \times \Sigma^r$, so that each state is of the form $((x_1, ..., x_{2r}), (y_1, ..., y_r))$. Thus each state in the automaton represents all of the context necessary for one application of the local rule. The transition relation δ contains the relation $(q_1, (a, b), q_2)$ if $q_1 = ((x_1, ..., x_{2r}), (y_1, ..., y_r))$, $q_2 = ((x_2, ..., x_{2r}, a), (y_2, ..., y_r, b))$, and $\rho(x_1, ..., x_{2r}, a) = b$. The paths traversable by following the transitions in the automaton thus represent all valid pairs of words x, y such that $x \xrightarrow{\rho} y$.

If the boundary conditions of ρ are two-way infinite, then T is a ζ-automaton with $I = F = Q$. In this case, the transition automaton recognizes any bi-infinite word which has an run that never fails. If the boundary conditions of ρ are one-way infinite, then T is a Büchi automaton with $F = Q$, and I is the set of states $q \in Q$ which are of the form $q = (0, 0, ..., a), (0, 0, ...b)$ and $\rho(0, ..., 0, a) = 0$. This accounts for the finite boundary conditions at the beginning of the configuration. If the boundary conditions of ρ are finite, then T is a finite automaton with initial states I as in the one-way infinite case, and final states $F \subset Q$ of the form $q = (a, 0, 0, ...), (b, 0, 0, ...)$ and $\rho(a, 0, 0, ...) = 0$. A cellular automaton with finite, cyclic boundary conditions, in which the first and last cells are adjacent, is modeled using a finite automaton with the condition that the first and last few inputs must remain acceptable under the local map ρ, instead of the assumed empty cells modeled by zeros above.

In all cases, the transition automaton scans the infinite word corresponding to a pair of configurations (referred to below as *tracks*), and recognizes the word if the second configuration is a successor of the first under the global map of ρ. An example of the basic transition automaton for the elementary cellular automaton described by Wolfram's Rule 30, assuming one-way infinite boundary conditions, is shown in Figure 3.1. This particular cellular automaton is of interest because it exhibits chaotic behavior, and has been proposed as a generator for pseudorandom numbers [23, 22].

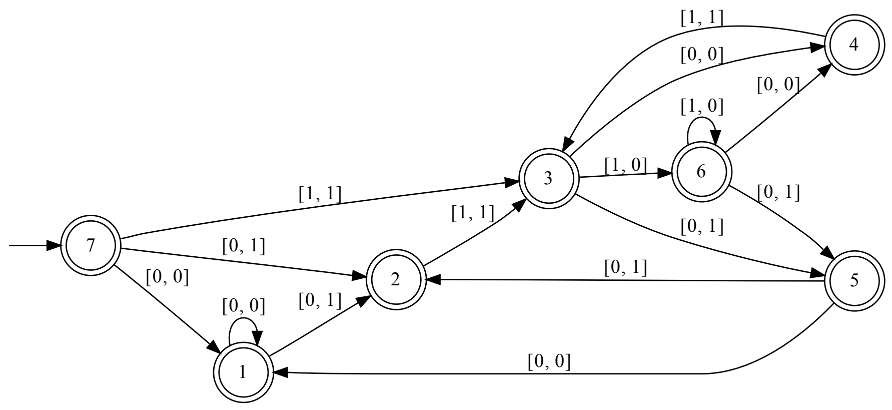

Figure 3.1: A basic transition automaton for ECA 30.

3.2 Specification

Specification in model-checking is simply the process of stating the properties that a model or design must satisfy. In order to specify properties about a cellular automaton ρ, we construct a first-order structure for our logical theory with the phase-space of configurations and the binary predicate $\rightarrow$. The relation $A \rightarrow B$ in this logic is true if and only if $A \xrightarrow{\rho} B$.

Many naturally arising questions about cellular automata have to do with the evolution of various configurations under the automaton's global map. In order to answer these questions, we model them as assertions about the phase-space of the cellular automaton. A first-order logical structure can be constructed over the phase-space using the predicate of one-step evolution of configurations. Thus, a question such as "is there a fixed point?" can be translated to $\exists X : \; X \rightarrow X$. More difficult questions such as "is there a three-cycle?" will require the notion of equality or inequality as well:

$$\exists X, Y, Z : \ (X \to Y) \wedge (Y \to Z) \wedge (Z \to X) \wedge (X \neq Y)$$

3.3 Verification

To verify assertions, we build the corresponding ω- or ζ-automata and check them for emptiness. Emptiness of an automaton indicates that no accepting paths exist, and can be checked in linear time on the size of the automaton using depth-first search. If no accepting paths exist, it is clear that the assertion being modeled by the automaton is false. If accepting paths do exist, they are witnesses for the properties in question.

We have already shown how to construct the transition automaton to check the atomic predicate $\to$. We will build the automata constructing to more complicated formulae inductively, in a manner corresponding to the construction of those formulae.

To check the equality relation, an automaton of one state suffices. This machine, shown in Figure 3.2, simply checks that the character in each word is equal at every index. The inequality automaton is almost as trivial; shown in Figure 3.3, it requires two states and checks that there is some index where the characters in the two words differ. These automata function in either the one-way infinite or bi-infinite case.

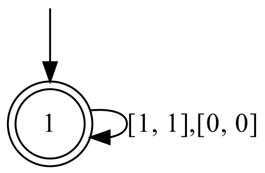

Figure 3.2: An automaton checking that two configurations are equal.

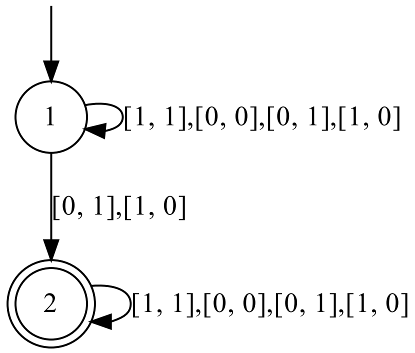

Figure 3.3: An automaton checking that two configurations are not equal.

Given two machines A_ϕ and A_ψ modeling logical formulae ϕ and ψ, we can construct a new automaton to model $(\phi \vee \psi)$ by taking the disjoint sum of A_ϕ and A_ψ. Since these automata may be nondeterministic, this is as simple as renaming the states of ψ to avoid intersection with those of ϕ. Similarly, the conjunction $(\phi \wedge \psi)$ of two formulae can be modeled by taking the product of the machines A_ϕ and A_ψ. An example of this product construction, the automaton which represents the formula $(X \rightarrow Y) \wedge (X \neq Y)$, is shown in Figure 3.4.

A new issue presents itself here: if we try to produce the formula for $(X \rightarrow Y) \vee (Y \rightarrow Z)$, the first tracks in each machine do not correspond to one another. We deal with this issue by maintaining a list which indicates which configurations correspond to which tracks in each automaton. When we operate on two automata with different sets of tracks, the new automaton scans words with a larger number of tracks, such that each configuration is treated appropriately. This allows us to intelligently merge, for example, the two tracks corresponding to configuration Y in the example above. The automaton which results is presented in Figure 3.5.

Existential quantifiers are handled by *projection*: erasing the track cor-

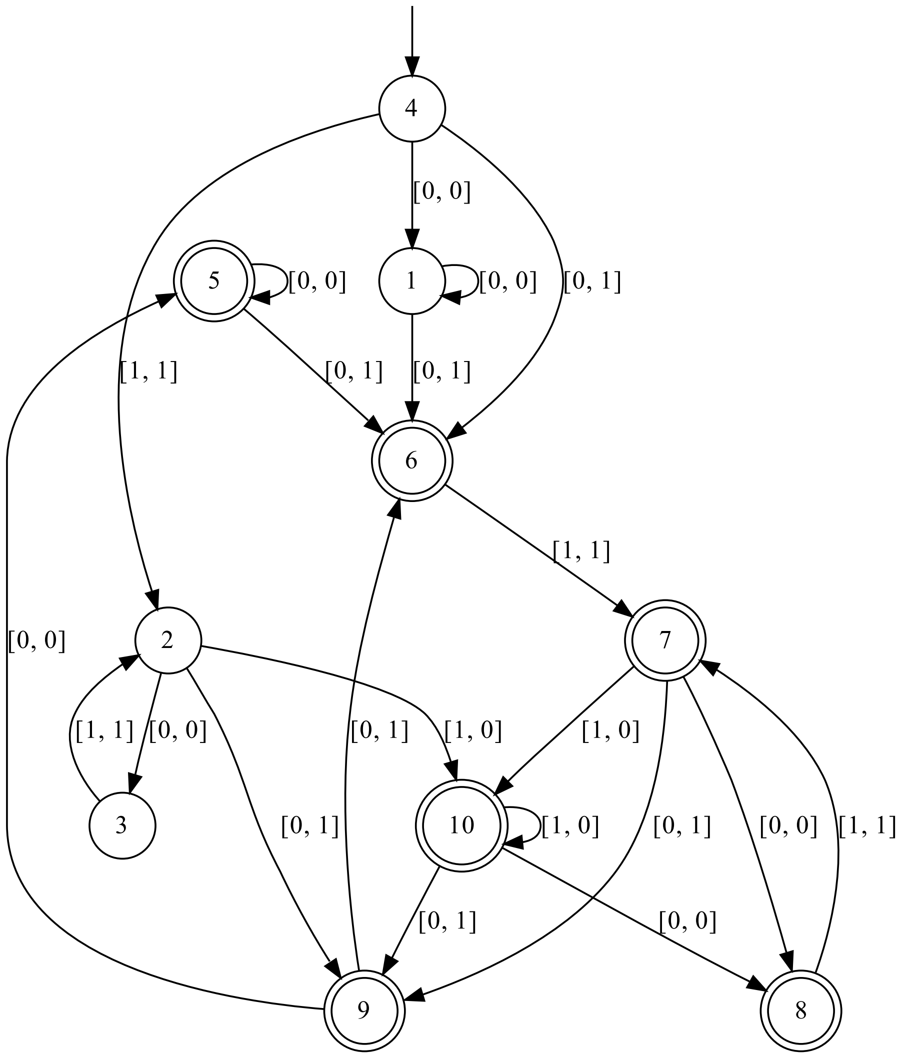

Figure 3.4: An automaton checking $(X \to Y) \land (X \neq Y)$ for ECA 30.

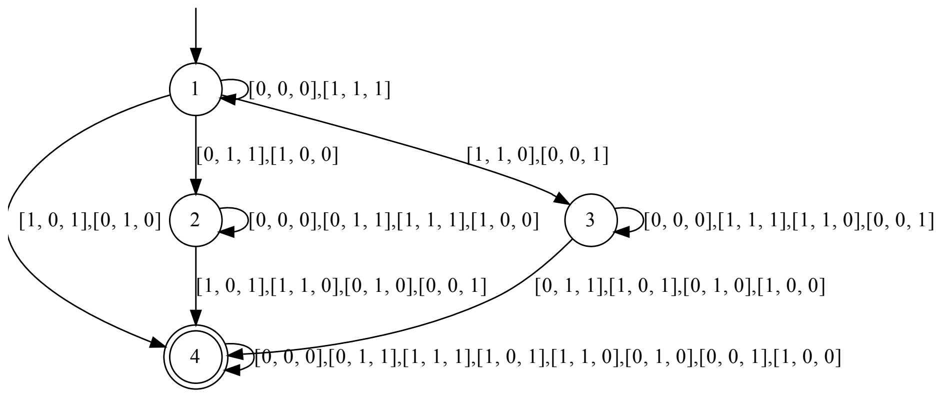

Figure 3.5: An automaton checking $(X \rightarrow Y) \vee (Y \rightarrow Z)$ for ECA 30.

responding to the variable being bound. Conceptually, this is because we no longer care what particular characters comprise the word in that track, as long as there exists some possible sequence of characters which would allow an accepting run. To handle universal quantifiers, we convert them to existential quantifiers.

Negation of a logical formula is simply complementation of the corresponding automaton, as described in Sections 2.2.2 and 2.2.3. Because of the exponential and super-exponential constructions involved in complementation, it represents the most expensive operation in terms of running time and the size of the automaton generated. Universal quantifiers can thus be doubly expensive: converting the expression $\forall Y\ \exists X : X \rightarrow Y$ to the equivalent form $\neg\ \exists X : \neg(\exists Y : X \rightarrow Y)$ may require us to perform complementation twice.

We are now prepared to verify assertions about the behavior of one-dimensional cellular automata, although our casual use of exponential and super-exponential algorithms indicates that computational feasibility may be a concern.

Chapter 4

Implementation of Safra's Construction

Considering the high-profile role that complementation plays in our model-checking procedure, an efficient algorithm is extremely important. Of the two algorithms for complementation discussed in Section 2.2.2, we chose to rely primarily on the determinization-based algorithm using Safra's construction (this design decision is explained in Chapter 5). We implemented Safra's construction as a standalone package, so that future work on ω-automata can easily reuse our implementation of this operation. Our intention is to make this implementation, as well as the Java code from the model-checking system, available under the GNU General Public License.

The running time and memory usage of the determinization algorithm are obviously related to the size of the generated Rabin automaton. Since the maximum size of the automaton generated by determinization is $2^{O(n \log n)}$, it is clear that any optimizations which can be made are critically important. We present several such optimizations to Safra's construction below.

4.1 Improvements

A number of implementations of Safra's construction are presented in the literature. While some, such as [1], present incremental improvements to the construction, our findings indicated that larger improvements were necessary to keep the resulting automata within a size that would make model-checking feasible.

First, as specified in our definitions in Section 2.1.2, we allow our automata to be incomplete. If not every transition needs to be defined, this allows a massive reduction in the number of transitions stored in memory. This also acts to simplify further operations such as product constructions.

4.1.1 Transition Ordering

The order of the first two steps of Safra's construction may be exchanged without affecting the proof of correctness presented in [15]. These steps represent constructing a new child of each node and transitioning the label of each node according to the behavior of the nondeterministic automaton. Previous definitions of Safra's construction have constructed new children first, and then transitioned the label of each node.

Our implementation provides an option to reverse the order of these steps. After finding that it typically reduces state complexity in the automata generated during the model-checking procedure, we use this option as a default. Transitioning the labels of each node first often reduces the number of states in the label, leading to a smaller number of trees and creating a smaller Rabin automaton.

4.1.2 Marking New Nodes

As a further optimization, new nodes which are created in step 2 of Safra's construction should be marked. This is also compatible with the proof of correctness of the algorithm. Informally, a marked node represents a recurrent path intersecting the set of final states. Since the label of a new node is a subset of the final states, it represents such a path. The traditional strategy where a new node is not marked, may require a number of additional transitions. Eventually this node will become marked in step 5 of Safra's construction, but the additional transitions to reach that point require additional time and increase the number of states in the automaton.

4.1.3 Advanced Node Renaming

Finally, an advanced heuristic for naming new nodes can be employed to further reduce the size of the resulting Rabin automata. Safra's construction algorithm proceeds until all states have been fully explored, so transitions leading to a previously seen state should be preferred over those leading to a new, equivalent state. When a new node is created, we check all possible labels for that node to see if using any of them will make it more likely that we will generate a previously-seen tree. This has the advantage of creating the fastest possible return to a previously seen state, which accelerates the process of Safra's construction.

The disadvantage of this heuristic is the increased time required to check all node labels. Since there are $2n$ possible labels for a node in the tree, and some number of nodes are created simultaneously in each step, an intuition might be that checking for a previously seen tree requires $\Omega(2^n)$ time at each step. However, we note that the default node-naming strategy is "use the lowest available node name." The only times when we should deviate from this strategy, therefore, is when there is an already-seen tree with an unused node ID less than its maximum node ID. The node renaming heuristic thus reduces to storing a reference to each of these trees and renaming the missing nodes appropriately during transitions of trees on the appropriate size.

Looking for existing trees that satisfy the condition, checking structural similarity, and renaming the nodes appropriately can be done in a time proportional to the size of the trees. This cost is asymptotically insignificant compared to the benefit of reducing the size of the automaton.

4.2 Performance Analysis

Of critical interest is the question of whether our improvements make any substantial difference to the performance of Safra's construction. We argue that they do through presenting example determinizations and statistics about the determinization of large numbers of ω-automata. These are also useful in grasping the time and memory requirements of our algorithms.

4.2.1 Sample Determinizations

As a first example of the improvements to Safra's construction defined above, we demonstrate different ways to determinize a simple Büchi automaton. Our example is the automaton presented in Figure 2.3, which recognizes the language of strings over $\{a, b\}$ with only finitely many a's. Any combination of our optimizations produces a correct automaton, so these machines are all equivalent. Figure 4.1 demonstrates the unoptimized result of Safra's construction that is presented in [1]. In Figure 4.2, we demonstrate the result of marking new nodes when performing the same determinization, and Figure 4.3 demonstrates the use of all our optimizations (except for advanced node renaming) in the construction.

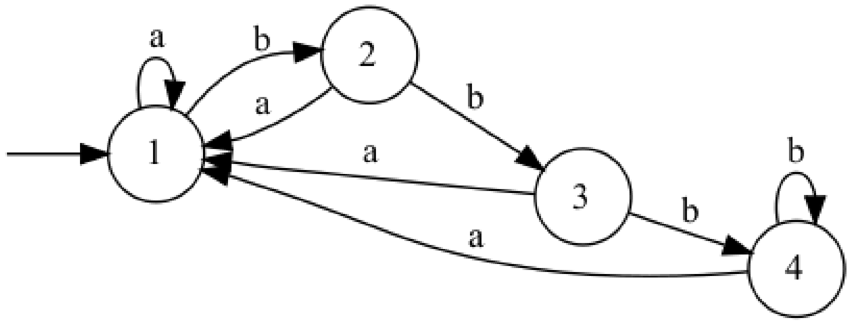

Figure 4.1: Rabin automaton equivalent to the Büchi automaton in Figure 2.3, produced by the unmodified version of Safra's construction. The acceptance condition for this automaton is $\{(\{1,2\},\{4\})\}$.

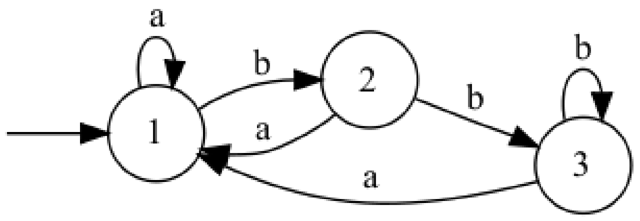

Figure 4.2: Result of marking new nodes when determinizing the automaton from Figure 2.3. The acceptance condition for this automaton is $\{(\{1,2\},\{3\})\}$.

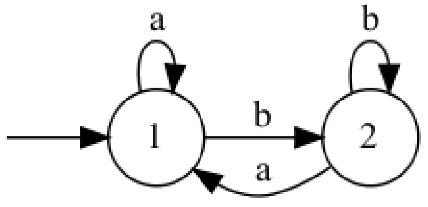

Figure 4.3: Optimal Rabin automaton equivalent to the automaton from Figure 2.3, produced using the optimizations to Safra's construction described in Section 4.1. The acceptance condition for this automaton is $\{(\{1\},\{2\})\}$.

Automata produced by determinization are not always so well behaved. Figure 4.4 illustrates a Büchi automaton of four states over the alphabet $\Sigma = \{1, 2, 3, \#\}$ presented in [1]. The determinization of the automaton with none of our optimizations results in a Rabin automaton of 384 states, shown in Figure 4.5.

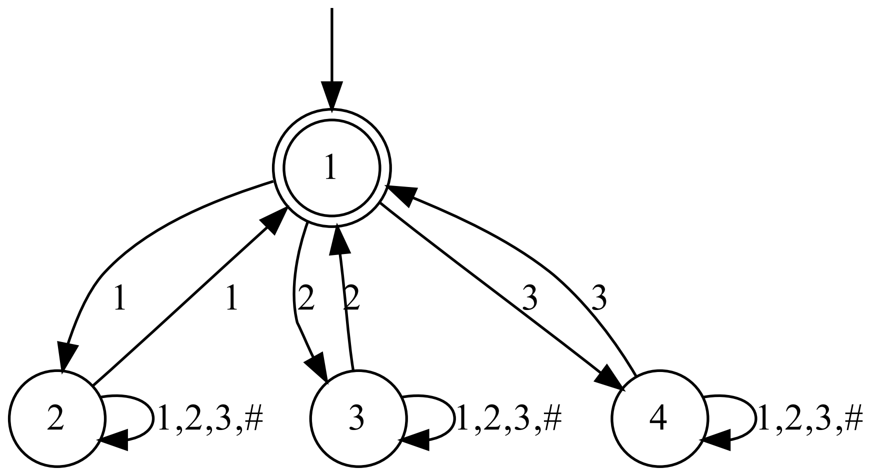

Figure 4.4: A Büchi automaton which exhibits an exponential blowup in state size under determinization. The results of determinization without and with optimization are shown in Figures 4.5 and 4.6, respectively.

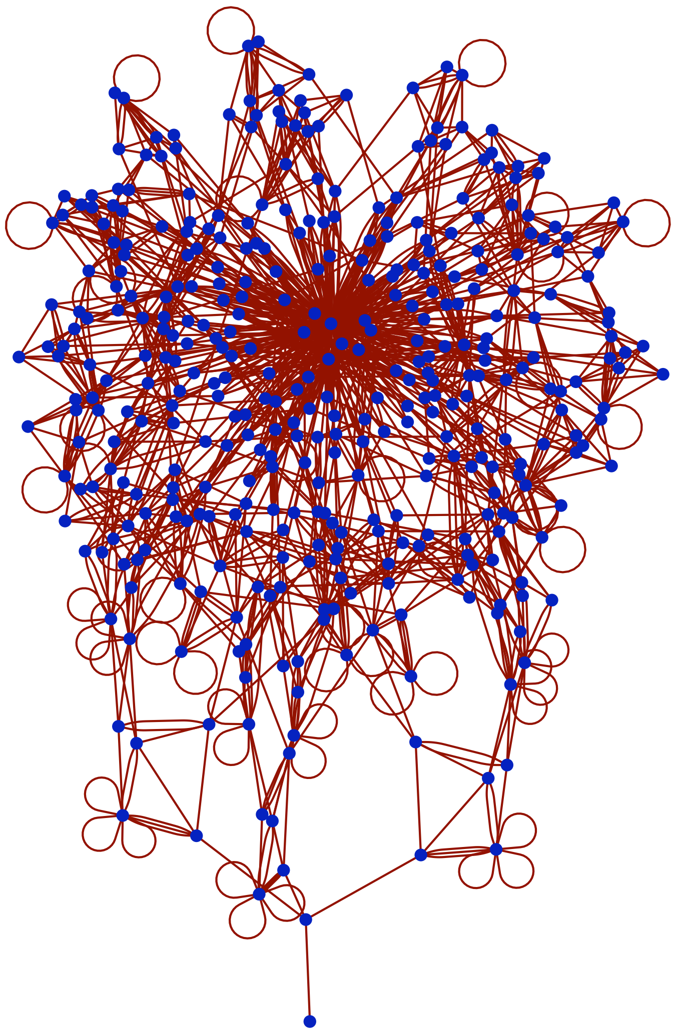

Figure 4.5: The transition matrix of the Rabin automaton resulting from the determinization of the automaton in Figure 4.4; it contains 376 states.

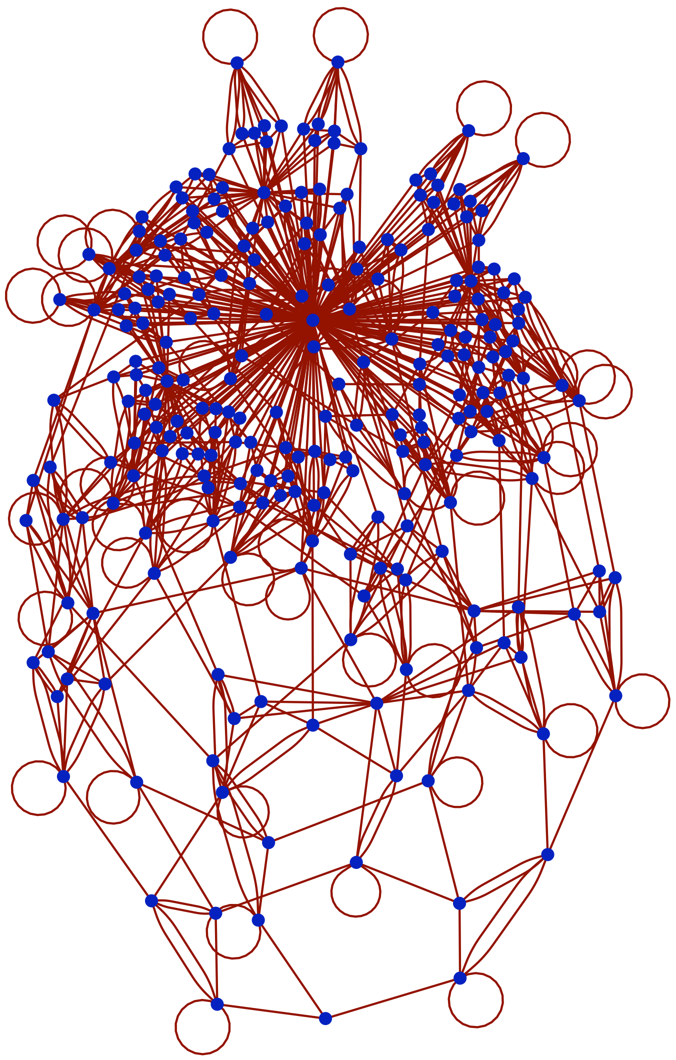

Figure 4.6: The transition matrix of the automaton resulting from the determinization of the automaton in Figure 4.4 if our optimizations are used; it contains 256 states.

As dramatic as the increase in state size caused by that determinization is, worse examples abound. We once again turn to [1], this time for an automaton which blows up to a total size of 13696 states under classical determinization. With our optimizations, this is reduced to 10776 states, but this is a hollow victory given that the original size of the automaton is only 5 states, and the alphabet $\Sigma = \{1, 2, 3, 4, \#\}$ contains only five characters. The automaton is presented in Figure 4.7, the unoptimized determinization in Figure 4.8, and the optimized determinization in Figure 4.9.

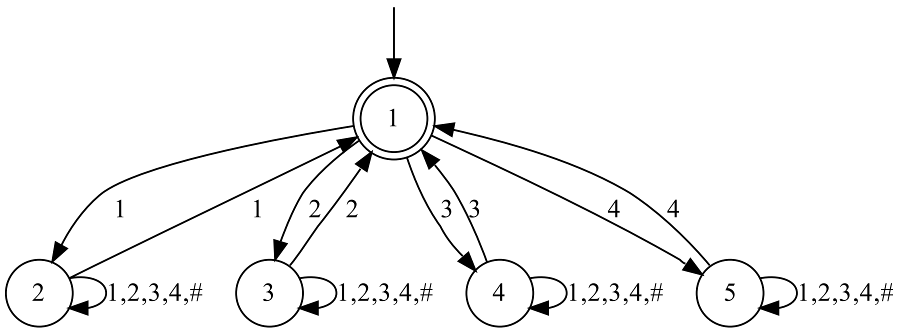

Figure 4.7: Another Büchi automaton exhibiting exponential blowup in state size under determinization. Determinization results are given in Figures 4.8 and 4.9.

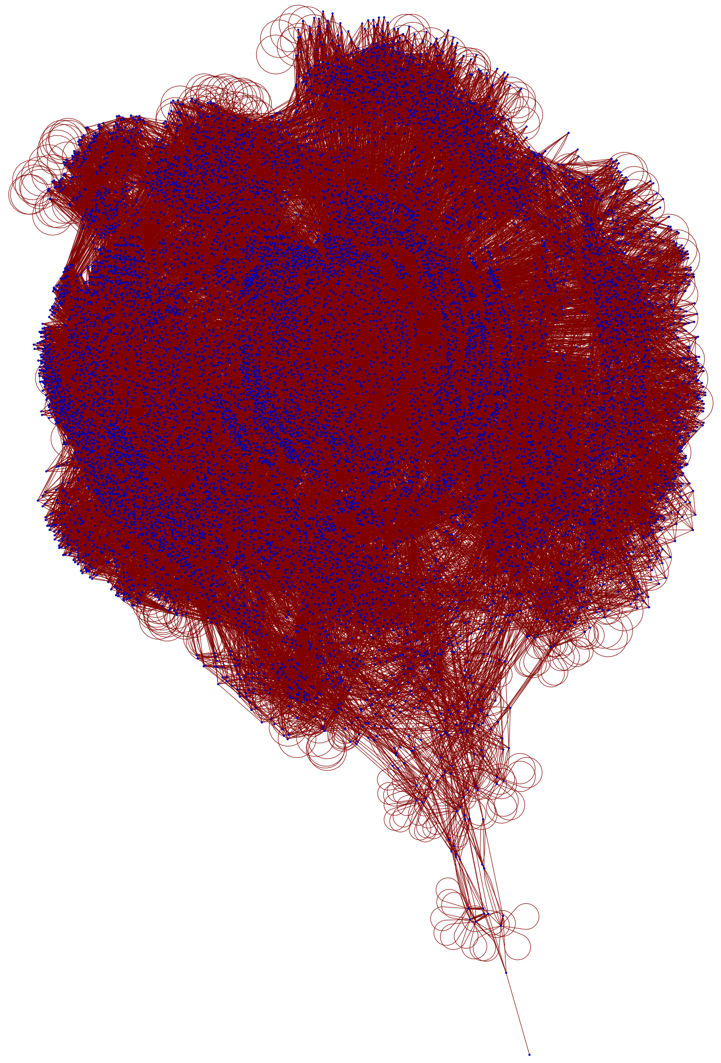

Figure 4.8: The transition matrix of the Rabin automaton resulting from the determinization of the automaton in Figure 4.7; it contains 13696 states. Note in particular the structure to the bottom of the diagram, which includes the initial state and a sink state.

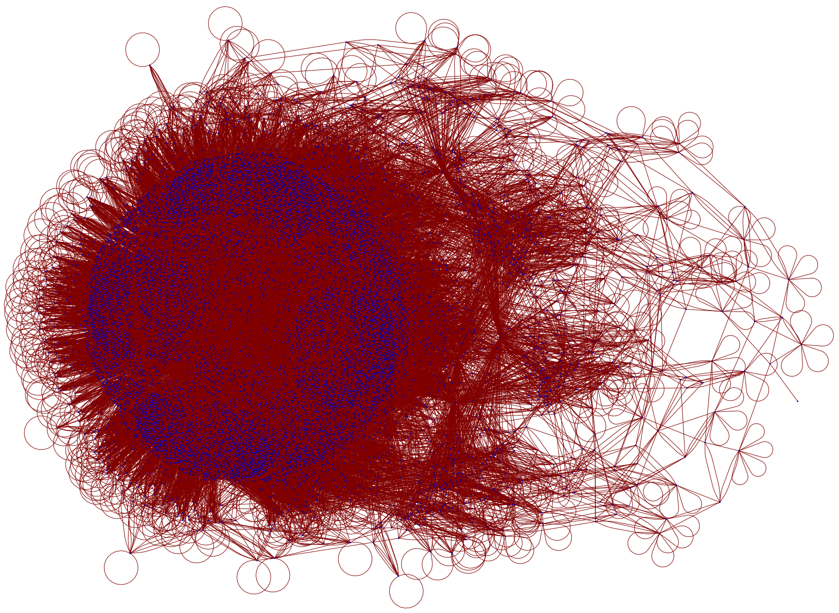

Figure 4.9: The transition matrix of the automaton resulting from the determinization of the automaton in Figure 4.7 if our optimizations are used. It contains 10776 states but nonetheless represents an improvement over the unoptimized determinization.

4.2.2 Performance Analysis

To further characterize the performance of Safra's algorithm and our improvements in the determinization of Büchi automata, we present statistics on the performance of both algorithms the determinization of randomized Büchi automata. There is some discussion as to the correct means of generating a "random" automaton. Our first method of obtaining a sampling creates a random graph, while a method that proved more useful in examining performance was to sample the automata generated by the model-checking engine.

A random sampling of Büchi automata is obtained by creating a random graph, and varying the parameters of size and *average connectivity*. When generating a random automaton of size n over an alphabet Σ of size k, there are a possible $\frac{(n^2)(k)}{2}$ transitions. To generate a random automaton, we begin with an empty transition relation $\delta = \emptyset$, and add each of these transitions to δ with probability $c \in [0, 1]$. We describe c in this section as the average connectivity of the automaton; note that this is semantically distinct from the concept graph connectivity.

As a first sample, we processed 100 random Büchi automata from size 1 to 8 and connectivity 0.1 to 0.9. Our findings, detailed in Tables 4.1 and 4.2, seemed to indicate that our optimizations yielded a slightly better performance in some cases and slightly worse performance in other cases. We present the average number of states in the Rabin automata resulting from determinization with and without our optimizations below.

Table 4.1: Average size of Rabin automata produced by the unoptimized determinization of Büchi automata of varying size and average connectivity.

Büchi States	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	3	3	3	3	3	2
3	3	9	13	16	15	13	11	7	5
4	8	41	75	68	49	28	15	10	6
5	20	375	530	211	81	39	19	11	6
6	91	1101	2531	270	104	42	19	11	7
7	259	5792	4519	532	75	36	18	12	7
8	1920	38396	2756	249	68	33	20	12	7

Table 4.2: Average size of Rabin automata produced by the optimized determinization of Büchi automata of varying size and average connectivity.

Büchi States	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	3	3	3	3	3	3
3	3	8	14	17	13	13	10	8	5
4	8	31	82	73	54	29	16	9	5
5	20	250	378	214	78	39	19	11	6
6	83	1416	1785	369	91	40	20	11	7
7	391	7165	3034	300	82	35	19	12	7
8	844	41591	2493	267	74	33	19	12	7

However, further investigation indicated that this average-case performance was misleading. First, the random-graph method of generating an automaton is not a good sampling: this machine is frequently empty or trivial. This reduces the number of samples which undergo significant processing, so there is an increased amount of variance in the data. Furthermore, the machines thus constructed are not representative of automata generated during the model-checking procedure.

In order to generate a more representative sample, we used the machines generated when checking for injectivity in each of the elementary cellular automata (see Section 5.2.2 for details of this procedure). Examining the effect of our optimizations in determinization of these machines, we found that the typical result was a 10-15% reduction in the size of the resulting automaton. The results are shown in Figure 4.10. While it remains premature to comment on the efficiency of our methods for all Büchi automata, we may be reasonably confident in their utility for our purposes.

Finally, a characterization of the performance of our implementation of Safra's construction would be incomplete without a discussion of time and memory usage. A good example automaton for stress-testing our implementation, is an automaton also exhibiting superexponential blowup. Presented in [20], it is shown in Figure 4.11. The Rabin automaton generated from determinization of this automaton contains 979,314 states if our optimizations are used, or 1,107,016 states if they are not.

As a relative benchmark, this automaton requires approximately 3 minutes 26 seconds to generate on a 2.4 GHz Intel Core 2 Duo processor. It uses approximately 1308 megabytes of RAM during this procedure. Since this automaton is significantly larger than any we have encountered during model-checking, it is reasonable to be optimistic about the performance of our algorithm. However, this level of memory usage could potentially be an issue in performing operations on multiple automata, and future improvements to the model-checking engine should also work on reducing memory overhead.

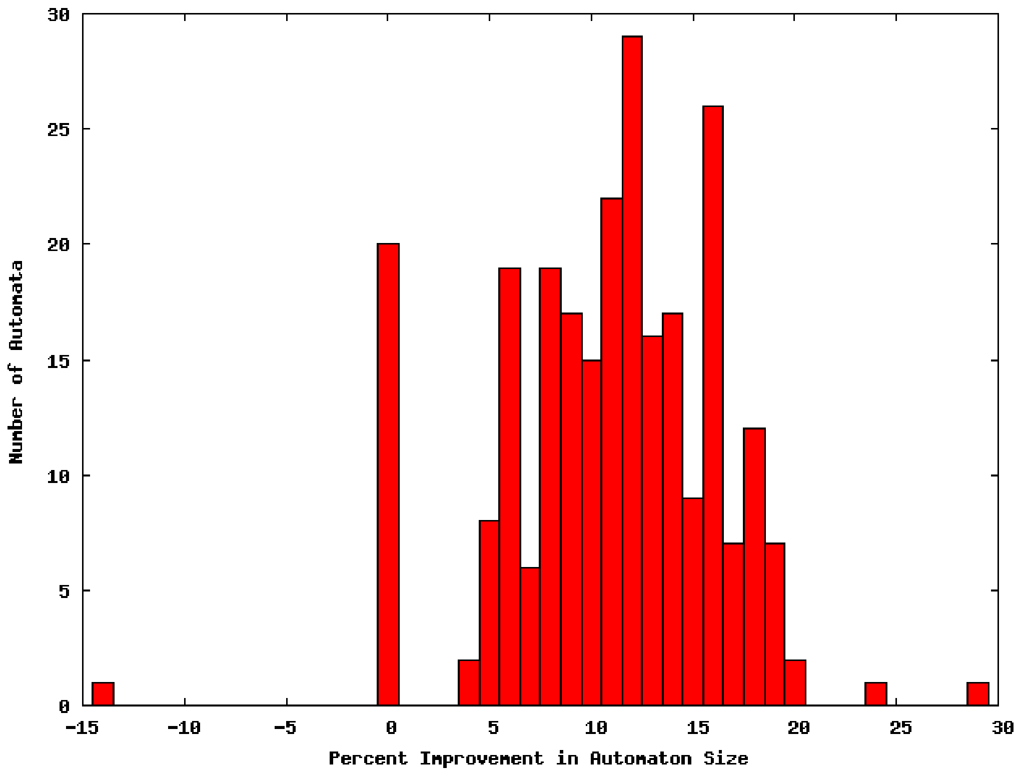

Figure 4.10: Effects of our optimizations to Safra's construction in reducing the size of automata produced by the model-checking system.

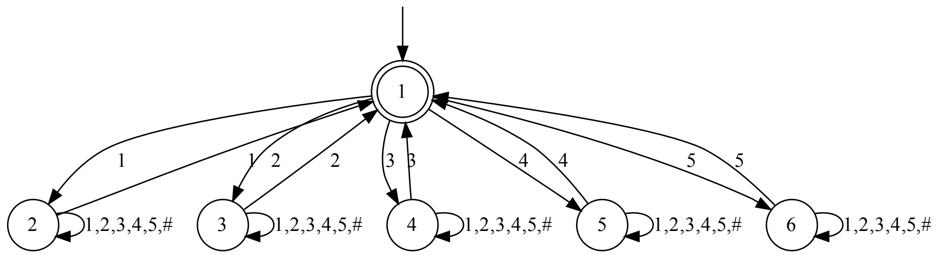

Figure 4.11: This Büchi automaton produces a Rabin automaton of 979,314 states when determinized.

Chapter 5

Model-Checking Engine

The implementation of our model-checking system is written primarily in Java, with an interface in Mathematica. Using Mathematica's JLink capability, we combine the advanced user interface and parsing features of Mathematica with a more powerful, efficient back-end written in Java.

5.1 Design Decisions

5.1.1 Implementation Languages

A major design decision in the implementation of our model-checking system was the use of Mathematica for the user interface and parser and Java for the back-end implementation of the algorithm. The decision to use Mathematica was influenced by the natural ease of parsing logical formulae in a functional programming language, as well as the fact that Mathematica is frequently used in the study of cellular automata.

Java suggested itself as a programming language when it became clear that an efficient implementation of Safra's construction and of the model-checking utility would require the heavy use of hashtables. A more important reason for the use of Java, however, is the use of Mathematica's JLink interface. This allows for extremely easy integration of the two components, and the efficient passing of data between the user interface and the high-performance back-end.

5.1.2 Complementation Algorithm

Also deserving of explanation is our decision to use an algorithm based on explicit determinization and Safra's construction for the complementation of ω-automata, rather than a method based on direct complementation. Safra's construction is an iterative process which generates the accessible part of a Rabin automaton based on exploration of the graph formed by the state set and transition relation. In contrast, the method for direct complementation presented in [17] creates the entire automaton, including inaccessible parts, and requires referencing and performing ranking operations on a set of objects of size $O(n^n)$.

In the worst case scenario, complementation of the Büchi automaton via determinization and Safra's construction will produce a larger automaton than direct complementation. However, we know that complementation of Büchi automata, even via the direct method, has a lower bound of $O((0.76n)^n)$ in the worst case. The best known algorithm generates an automaton of $O(n^2(0.76n)^n)$ states on an input of n states. Thus, even a Büchi automaton of 10 states can generate an automaton of 6,428,888,930 states. Because it the complementation method does not iteratively construct the graph, an implementation may generate this many states even in cases where the the majority of the graph is inaccessible. We therefore use Safra's construction and the associated complementation algorithm, and resign ourselves to the fact that the problems where direct complementation outperforms our algorithm are beyond our current computational resources.

5.2 Usage

The use of the model-checking software is best explained via examples. Simple properties such as surjectivity and injectivity are of interest to those studying cellular automata, but also simple enough in their formulation to make ideal candidates for model-checking.

5.2.1 Checking Surjectivity

A given automaton ρ is *surjective* if every configuration C has at least one predecessor P such that $P \xrightarrow{\rho} C$. In the theory of phase-space, this property can be expressed as

$$\forall x \; \exists y : y \to x$$

Our Mathematica parser translates this formula into an expression that is decidable in the model-checking system. This is accomplished by replacing any universal quantifiers with appropriate existential quantifiers,

$$\neg\exists x\ \neg\exists y : y \rightarrow x$$

parsing the components of the formula appropriately,

$$\neg(\exists x(\neg\exists y(y \rightarrow x))))$$

and converting the resulting formula into a series of operations to pass via JLink to the model-checking functions:

```
setCA[ρ];

isEmpty[
  project[ x,
    not[
      project[ y,
        step[y, x]
      ]
    ]
  ]
];
```

The preamble `setCA` sets the appropriate fields in the parser to allow us to work with the automaton ρ. The notation `step` signals for construction of the basic transition automaton, while the syntax `project[y, A]` calls for erasing the track corresponding to `y` from automaton `A`. The operator `not` calls for complementation.

It is important to note why, instead of complementing the entire automaton, we made a call to `isEmpty`. The automaton passed to `isEmpty` has had all tracks erased. If it is empty, this indicates that there are no witnesses for the underlying proposition. Otherwise, every accepting run corresponds to a witness. It is true that we could complement the automaton and check for universality. However, emptiness for Büchi automata can be checked in linear time via depth-first search, so the most efficient way of evaluating the expression is to check for emptiness.

If the automaton is empty, then there are no witnesses for

$$\exists x\ \neg\exists y \rightarrow_{\rho} (y, x)$$

or we know that there does not exist a configuration in the phase-space which lacks a predecessor. Therefore, we can safely conclude that if this Büchi automaton is empty, the cellular automaton ρ is surjective. Note also that if the automaton had not been empty, the depth-first search process which tested emptiness could also produce a list of witnesses.

In the particular case of surjectivity, the counterexamples would be "Garden of Eden" configurations. These are configurations for which no predecessor exists. The construction of Garden of Eden configurations has been an area of active study with respect to cellular automata (additional detail is provided in [10, 16]), so generating such configurations automatically is one possible use of our model-checking application.

When we ran this algorithm on each of the elementary cellular automata with one-way infinite boundary conditions, we determined that the only surjective elementary cellular automata are those characterized by Wolfram's rules 51, 60, 85, 86, 89, 90, 101, 102, 105, 106, 149, 150, 153, 154, 165, 166, 169, 170, 195, and 204.

5.2.2 Checking Injectivity

A cellular automaton is injective if every configuration has at most one predecessor. The corresponding assertion can be expressed as in our theory as

$$\neg\ \exists X, Y, Z : (X \to Z) \wedge (Y \to Z) \wedge (X \neq Y)$$

The parser translates this formula into the following expression:

$$\neg\ (\exists x\ (\exists y\ (\exists z : (x \to z) \wedge (y \to z) \wedge (x \neq y)))))$$

We can see that the series of commands given to the model-checking is slightly more complicated than when checking surjectivity:

```
setCA[ρ];

isEmpty[
 project[x,
   project[y,
    project[z,
      and[
        and[
          step[x, z],
          step[y, z]],
        unequal[x, y]
      ]
    ]
  ]
];
```

The new notation `and` simply signals for construction of the product automaton of its two arguments. We also have an operation `or` which constructs the disjoint sum of its arguments.

If the automaton is empty, then there are no witnesses for

$$\exists X, Y, Z : (X \to Z) \wedge (Y \to Z) \wedge (X \neq Y)$$

so we know that no two configurations in the phase-space map to the same configuration under the global map of the cellular automaton. Therefore, the corresponding cellular automaton ρ is surjective.

When we ran this algorithm on each of the elementary cellular automata with one-way infinite boundary conditions, we determined that the only injective elementary cellular automata are those characterized by Wolfram's rules 15, 51, 60, 195, 204, and 240. As an aside, this implies that rules 51, 60, 195 and 204 are bijective.

Chapter 6

Extensions

Many significant improvements are possible to our model-checking system. In this chapter we discuss several possible extensions to the system which might greatly improve its performance or capabilities.

6.1 Improving Complementation

As discussed briefly before, methods of direct complementation offer the potential for significantly improving the worst-case runtime and size complexity of complementation. The asymptotically best known algorithm, presented in [17], is within $O(n^2)$ of the lower bound on complementation. However, as we discussed in Section 5.1.2, a method which constructs only the accessible portion of the automaton is critical for our purposes. Any work on Schewe's algorithm which allowed this could be translated into a significant improvement in the efficiency of our model-checking algorithms. Work to avoid explicit determinization via the use of antichains, presented in [5] and [3], also represents a significant potential improvement in algorithmic efficiency and should be investigated in this context.

Some improvement may be possible even without additional theoretical work, however. An algorithm for simultaneous determinization and complementation of ω-automata is presented in [6]. Implementing the algorithm developed by Emerson and Jutla in place of Safra's construction has the potential to improve our constructions by an exponential factor, and would require only the additional step of converting Rabin automata back to equivalent Büchi automata. Additionally, it would be worthwhile to test the average-case performance against Safra's construction to ensure that the worst-case exponential improvement proven by Emerson and Jutla also extends to typical examples.

6.2 Parallelization

With recent advances in high-performance parallel computing, it is natural to ask whether any additional mileage can be obtained from parallelizing the algorithms involved in our model-checking system. The obvious candidate for parallelization is the complementation of Büchi automata because of the massive cost involved. Another task which could from parallelization might be the complementation of ζ-automata, since multiple complementations of ω-automata can be performed simultaneously in that process. However, the number of simultaneous complementations in this procedure is several orders of magnitude less than the number of states explored during one complementation of an ω-automaton.

The iterative determinization of a Büchi automaton is essentially a process of graph exploration. This can be modeled by depth-first search or breadth-first search on the state set. Some relevant work by Barnat et al. in [2] explored an algorithm for parallelization of breadth-first search with respect to model-checking problems in linear temporal logic. An adaptation of this algorithm could be extremely valuable in accelerating parallelization. The critical obstacles to overcome in any such algorithm are the synchronization of data between multiple processors and the distribution of work. In this respect, it may be possible to exploit the underlying structure of the Büchi automata to improve the performance of parallelization. As an example, consider the Büchi automaton in Figure 6.1. This automaton recognizes the language $(a + b + c)^*((a + b)^\omega + (b + c)^\omega + (a + c)^\omega)$, or the set of all one-way innite words over $\{a, b, c\}$ with nitely many as, nitely many bs, or nitely many cs. The result of determinization, shown in Figure 6.2, indicates a natural strategy for parallel graph exploration. The resulting Rabin automaton consists of a root node, three intermediate nodes, and six strongly connected components, suggesting that each of the components could be explored in parallel, minimizing the need for synchronization and providing excellent distribution of work.

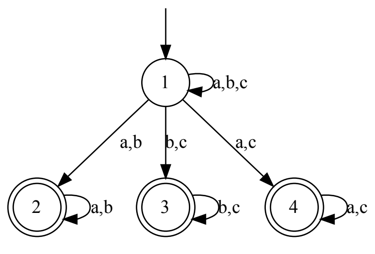

Figure 6.1: A Büchi automaton recognizing the language $(a + b + c)^*((a + b)^\omega + (b + c)^\omega + (a + c)^\omega)$.

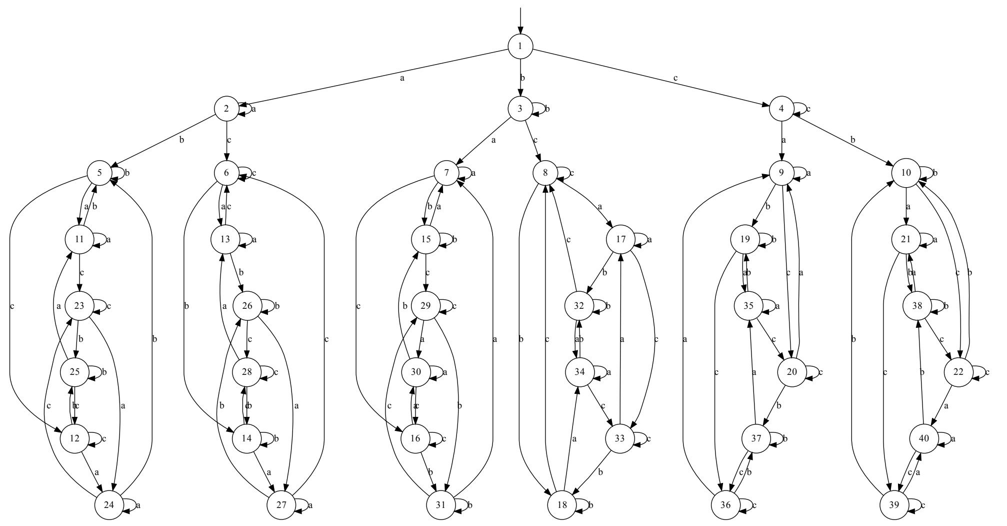

Figure 6.2: A Rabin automaton equivalent to the machine above. Note the large number of strongly connected components, implying a high potential for parallelization.

6.3 More Expressive Logics

In [19], it is established that the first-order logic we use to describe properties of the global map of cellular automata is decidable. It is left as an open question, however, whether the algorithm we use to decide these properties can be generalized and applied to more expressive logics. We know that some properties, such as the reachability of a given configuration, are undecidable [18] in the general case.

An extension to this theory is presented in [7] using results about ω-automatic structures. This shows that a logic containing counting and cardinality quantifiers is decidable for one-dimensional cellular automata. Using this logic, for example, we would be able to determine whether there were countably or uncountably many fixed points or cycles of given lengths. The extension of our model-checking system to include this logic would thus significantly increase the properties verifiable by the system.

Chapter 7

Conclusion

In this thesis, we have made several contributions towards the study of cellular automata and ω-automata. First, we have made significant progress towards demonstrating the feasibility of model-checking cellular automata. While our implementation is far from comprehensive, it clearly indicates that Sutner's procedure for model-checking cellular automata is amenable to practical use. Further work on this implementation should yield more information about the capabilities and limits of this technique with today's hardware limitations.

Second, we have provided a reference implementation for Safra's determinization algorithm, and demonstrated several optimizations for the construction. We are hopeful that this implementation could form the core of a common library for operations on ω-automata. Such a library, particularly if it included the future work discussed earlier in this thesis, could greatly facilitate the study of automata on infinite words. Automata on infinite words are being used more for model-checking various systems, and a common library for using these automata could aid researchers and engineers.

Finally, our implementation of the model-checking system is itself a useful tool for the study of cellular automata. We have shown how the system can already be used to check common properties such as injectivity and surjectivity, and discussed how it could be improved in several ways. With continuing work, this system should be capable of demonstrating important properties of cellular automata. It is our hope that further development work on the system will produce a valuable tool for students, researchers, and scientists who use cellular automata in their work.

Bibliography

[1] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier. Observations on determinization of Buchi automata. *Theoretical Computer Science*, 363(2):224 – 233, 2006. Implementation and Application of Automata, 10th International Conference on Implementation and Application of Automata (CIAA 2005).

[2] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *18th IEEE International Conference on Automated Software Engineering*, pages 106–115, 2003.

[3] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games of incomplete information. In *Proceedings of CSL 2006: Computer Science Logic*, Lecture Notes in Computer Science 4207, pages 287–302. Springer-Verlag, 2006.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.

[5] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proceedings of CAV 2006: Computer-Aided Verification*, Lecture Notes in Computer Science 4144, pages 17–30. Springer-Verlag, 2006.

[6] EA Emerson and CS Jutla. On simultaneously determinizing and complementing omega-automata. In *Fourth Annual Symposium on Logic in Computer Science*, pages 333–342, 1989.

[7] Olivier Finkel. On Decidability Properties of One-Dimensional Cellular Automata. *Equipe de Logique Mathematique*, 2009.

[8] Karel Culik II and Sheng Yu. Cellular automata, omega omega-regular sets, and sofic systems. *Discrete Applied Mathematics*, 32:85–101, 1991.

[9] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[10] Kenichi Morita. Reversible computing and cellular automata—a survey. *Theoretical Computer Science*, 395(1):101–131, 2008.

[11] M. Nivat and D. Perrin. Ensembles reconnaissables de mots biinfinis. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 47–59. ACM New York, NY, USA, 1982.

[12] Dominique Perrin and Jean-Eric Pin. *Infinite Words*. Elsevier, 2004.

[13] Nir Piterman. From nondeterministic Buchi and Streett automata to deterministic parity automata. *Logic in Computer Science, Symposium on*, 0:255–264, 2006.

[14] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[15] S. Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327, Oct 1988.

[16] Palash Sarkar. A brief history of cellular automata. *ACM Comput. Surv.*, 32(1):80–107, 2000.

[17] Sven Schewe. Buchi complementation made tight. *26th International Symposium on Theoretical Aspects of Computer Science*, 2009.

[18] Klaus Sutner. Classifying circular cellular automata. *Physica D*, 45(1-3):386–395, 1990.

[19] Klaus Sutner. Model checking one-dimensional cellular automata. *Journal of Cellular Automata*, 2007.

[20] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[21] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238, 1996.

[22] M.Y. Vardi. Buchi Complementation: A Forty-Year Saga. In *5th symposium on Atomic Level Characterizations (ALC'05)*, 2005.

[23] Stephen Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55(3):601–644, Jul 1983.

[24] Q. Yan. Lower Bounds for Complementation of omega-Automata Via the Full Automata Technique. *Lecture Notes in Computer Science*, 4052:589, 2006.