Model Checking Omega Cellular Automata

Joseph A. Gershenson

CMU-CS-10-123
May 2010

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the Degree of Master of Science.

Keywords: Automata theory, cellular automata, model checking, ω-automata

Contents

Abstract

The evolution of one-way infinite cellular automata can be described by the first-order theory of phase-space, which uses one-step evolution as its main predicate. Formulas in this logic can be used to express properties of the global map such as surjectivity. A complete implementation of the decision algorithm is provided, as well as methods for manipulating the Büchi automata on which the decision algorithm relies. Experimental results and a discussion of the tractability of larger problems are presented.

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 one-dimensional cellular automata.

The theory inspiring this paper is presented by Sutner in [34] 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 are built by combining one character from each of the two component words. The relation $A \to B$ corresponding to the global map can be checked using a substructure of the de Bruijn automaton over the new joint alphabet. This procedure may be generalized to check a conjunction of arbitrarily many predicates, and construction of more complicated formulas in the theory is done inductively by performing appropriate operations on the automata representing the appropriate sub-formulae.

A major contribution of this work is a complete implementation of the model-checking procedure in C++, including a library for manipulating Büchi automata independently of the model-checking algorithm. Additionally, we provide insight on the tractability of this problem and the degree to which some instances approach the theoretical bounds. The results of running the model-checking algorithm on several properties of elementary cellular automata are reported, as well as the efficiency and performance of the implementation itself.

As an extension of my undergraduate work, and in particular my senior honors thesis "Model Checking Cellular Automata," portions of this work necessarily appear derivative. There are, however, significant changes. The most prominent of these is the narrowing of focus to one-way infinite cellular automata, which is both indicative of the more thorough work in this thesis and a more accurate description of my research, since the undergraduate work did not completely implement model-checking for more than one-way infinite cellular automata either. The implementation of the model-checking algorithm presented here is completely distinct from my undergraduate work, and the experimental results shown in fact were not obtainable before due to the relative inefficiency of the previous implementation. Appropriately for a product of the Fifth Year Masters program, this also represents a much more complete and accessible extension of the undergraduate research. In particular, the model-checking implementation should be significantly easier for the interested reader to operate and modify.

The first chapters of this thesis constitute a review of definitions and a background of the problem, as well as a description of Sutner's model-checking algorithm. The latter portion is given over to a presentation of a complete implementation of the model-checking procedure and the presentation of experimental results. Finally, we discuss future work and the implications of this research for future study of model-checking cellular automata.

Chapter 2

Background

Modeling the evolution of one-way infinite cellular automata requires manipulating sets of infinite words. Here we review our working definitions of cellular automata and the ω-automata used to recognize these sets.

2.1 Definitions

2.1.1 Cellular Automata

A cellular automaton consists of an environment of cells and a rule which dictates their evolution over a series of discrete time steps. The contents of each cell are updated at each step depending on the concept of their neighbors. In the simplest configuration, cells are arranged in a one-dimensional line. More complicated environmental configurations are possible (additional information is given in [38] and [27]). Statements about the behavior of cellular automata quickly become undecidable if the environment is multidimensional, however, so in this work we consider only one-dimensional cellular automata.

Definition A *cellular automaton* is a local map $\rho : \Sigma^{2r+1} \to \Sigma$ 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 [38]; 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. Automata may also have a larger alphabet or a larger radius than in these examples; we will discuss in passing the generalization of our methods to working with automata of various alphabets and radii.

Another significant feature of a cellular automaton is the *boundary condition* of the environment. 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.

Alternatively, the boundary condition of a cellular automaton may be 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 words and languages: this motivates our introduction of ω-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 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 $G_\rho(u) = v$.

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 $x \overset{\rho}{\to} y \overset{\rho}{\to} z \overset{\rho}{\to} x$. Such a cycle in the cellular automaton is clearly equivalent to a 3-cycle in the phase-space. If the cellular automaton has finite boundary conditions, the phase-space is a finite graph, so we could directly check this property using a graph traversal algorithm such as breadth-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 build up this model-checking strategy in the following sections, and present an implementation as the main accomplishment of this thesis. Immediately, however, we review the definitions of the automata which recognize languages of infinite words.

2.1.2 Finite Automata

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. Excellent references for the interested reader can be found in [35] and [23]. To clearly define automata over infinite words, we first review the definitions of automata over finite words.

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 defines the basic transition system common to all automata.

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 is formalized using the concept of a *run*.

Definition A *run* of an automaton (Q, Σ, δ) over a finite word $w \in \Sigma^n$, where $w = w_0w_1...w_{n-1}$, 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 $q_n \in F$.

The concept of runs is used to formally define recognition of words. 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 2.1 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 common properties of automata. 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, this automaton is nondeterministic: not all transitions are unambiguous.

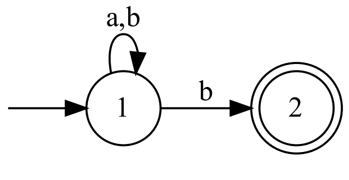

Figure 2.1: A simple finite automaton with alphabet $\{a, b\}$.

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 $|I| > 1$.

2.1.3 ω-Automata

Since modeling the evolution of one-way infinite cellular automata necessarily requires manipulating sets of infinite words, we here review definitions of the ω-automata used to recognize these languages. 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. Since there is no final character in an infinite word, it is also necessary to define a new acceptance condition for ω-automata.

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.

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.

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 $L(A) = \Sigma^\omega$.

An example of a Büchi automaton is given in Figure 2.2. 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 2.2 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, one can always construct a deterministic automaton which recognizes the same language as a nondeterministic automaton by using a power set construction (classically shown in [25]). However, the same is not true of

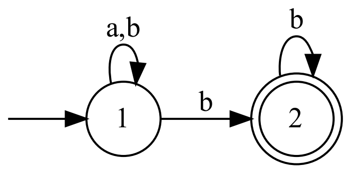

Figure 2.2: A simple Büchi automaton.

Büchi automata: there exist recognizable languages L such that L is not the language of any deterministic Büchi automaton. A formal proof, provided in [23], relies on the notion of prefixes. While recommended to the interested reader, it is beyond the scope of this thesis. The critical result is that a new type of ω-automaton must be used to generate 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 $r = q_0, q_1, ...$ 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 is presented in Figure 2.3. It recognizes the same language as the Büchi automaton of Figure 2.2: the set of strings over $\{a, b\}$ with only finitely many as.

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

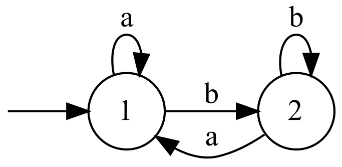

Figure 2.3: A simple Rabin automaton. The acceptance condition R is $\{(\{1\},\{2\})\}$, so an accepting run is one which reaches state 1 finitely often and state 2 infinitely often.

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 [25]: for every nondeterministic finite automaton, we can construct a deterministic finite automaton which accepts precisely the same language.

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

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

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 2.2. 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 McNaughton in [20]:

Theorem 2.1. *Any subset of Σ^ω which can be recognized with a Büchi automaton 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 [26, 23]. Safra's determinization algorithm is presented here for reference and because the algorithm features 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.

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 [37].

The best lower bound on the blowup of a Büchi automaton during complementation, established by Yan in [39], 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 reader is again referred to [37] and [28]. Two results in particular are worth mentioning here: an algorithm for complementation presented in [23] which makes use of Safra's determinization algorithm, and a direct algorithm for complementation recently presented in [28] 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. In this way the automaton effectively records which Rabin pairs are currently satisfied or unsatisfied, and the corresponding Büchi state may be added to the set of final states accordingly. We will avoid detailing the process further here, since we make no significant changes to this construction. The complete algorithm and proof of correctness is published in [23], and our implementation is available to be examined. Since the algorithm utilizes Safra's construction, and then builds additional copies of the automaton corresponding to satisfied Rabin pairs, the complementation procedure can generate an automaton of $2^{O(n \log n)} 2^{O}(n)$ states given an input automaton of n states.

The complementation method proposed by Schewe in [28], 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. Our motivation for using Safra's construction rather than this method is largely that Safra's construction is an iterative process. The determinization and complementation algorithm we use generates only the accessible part of a Rabin automaton based on exploration of the graph formed by the state set and transition relation. The method of [28] 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 having on the order of 6.4×10^9 states under the algorithm of Schewe. The upper bound on Safra's construction shows us the deterministic Rabin automaton generated could have up to 1.0×10^{10} states, and thus the complement Büchi automaton could have up to 1.0×10^{13} states. The saving factor of 0.76 introduced by Schewe becomes less relevant in practice in this example.

A large number of Büchi automata do not exhibit superexponential growth of the accessible part under complementation, however. By using Safra's construction and the associated algorithm to construct only the accessible part of an automaton for complementation, we keep many automata manageable afterwards. As we will demonstrate in Chapter 5, this allows checking surprisingly many simple formulas involving negation.

Chapter 3

The Model-Checking Algorithm

To answer questions about the behavior of cellular automata, we employ the strategy of model-checking. Model-checking, a method of formally verifying assertions about the behavior of systems, is described by Clarke et al. in [7] 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 research is primarily Sutner's constructive proof in [34] that model-checking for one-dimensional cellular automata is decidable. This chapter essentially presents an overview of the proof of this theorem using Clarke's phases of model-checking as a framework. Additionally, we adapt the general argument to model-checking omega cellular automata. The interested reader is also referred to the original exposition of the proof in [34]. 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 could be checked using a finite automaton. For our implementation, which examines cellular automata with one-way infinite boundary conditions, we use ω-automata. 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.

To model the evolution of one configuration A to another configuration B under ρ, we build the *transition automaton* T for the local map of ρ. This automaton accepts an infinite word if and only if the characters in words A and B are related by ρ. We can generalize this to an arbitrary number of predicates k by constructing the de Bruijn automaton over the alphabet Σ^k of order $2r$, where Σ, r are the alphabet and radius of ρ. This is actually a substructure of the complete de Bruijn automaton; in T transitions are only present if they are compatible with the labels in ρ. An infinite path in this automaton thus corresponds to a word satisfying $A \to B, C \to D, \ldots$ under ρ.

When 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 r inputs must be related 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 [38, 37].

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

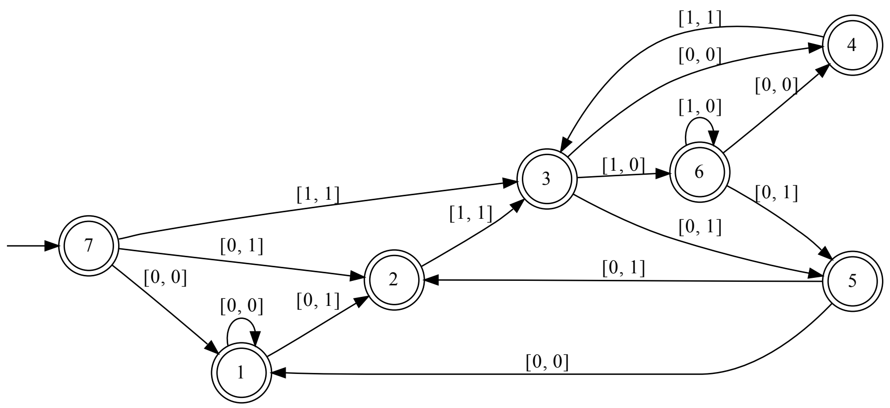

Figure 3.1: A basic transition automaton for ECA 30.

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:

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

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 $\rightarrow$. We add equality to our logic for no additional effort, since checking the formula $A = B$ can be done in precisely the same manner as checking $A \rightarrow B$. Because we are building a de Bruijn automaton over Σ^k anyway, it is easy to check a conjunction of these predicates. We build the automata constructing to more complicated formulae inductively, in a manner corresponding to the construction of those formulae.

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 3.2.

A new issue presents itself here: if we try to produce the formula for $(X \rightarrow Y) \vee (Y \rightarrow Z)$, for example, the first tracks in each machine correspond to different configurations and should not be conflated. Previously, we dealt with this issue by maintaining a list of which configurations corresponded to which tracks in each automaton. However, experimentation showed that it became more efficient to construct all transition automata directly over the maximum number of variables in the formula being modeled. This removes the need to maintain and update a dictionary of track / configuration semantics and increases the speed of product and sum construction. The automaton which results from the above formula is presented in Figure 3.3.

Via careful specification, many of the $O(n^2)$ product constructions that appear necessary can be avoided for formulas with a large number of conjunctions. We can encode a conjunction of literals $x \rightarrow y$ directly into a single transition automaton by including only the transitions where all literals are valid. Therefore, there is a tremendous advantage in writing the matrix of the formula we wish to verify in conjunctive normal form (CNF). We can also include negative literals of the form $\neg(x \rightarrow y)$ in the conjunction, but each such literal will double the size of the conjunction automaton. This is because a negated literal requires only one position at which the literal is unsatisfied; it is necessary to increase the state space of the automaton to record whether a given literal has yet been satisfied.

Negation of a logical formula is simply complementation of the corresponding automaton, as described in Section 2.2.2. Because of the exponential and super-exponential constructions involved in complementation, it represents the

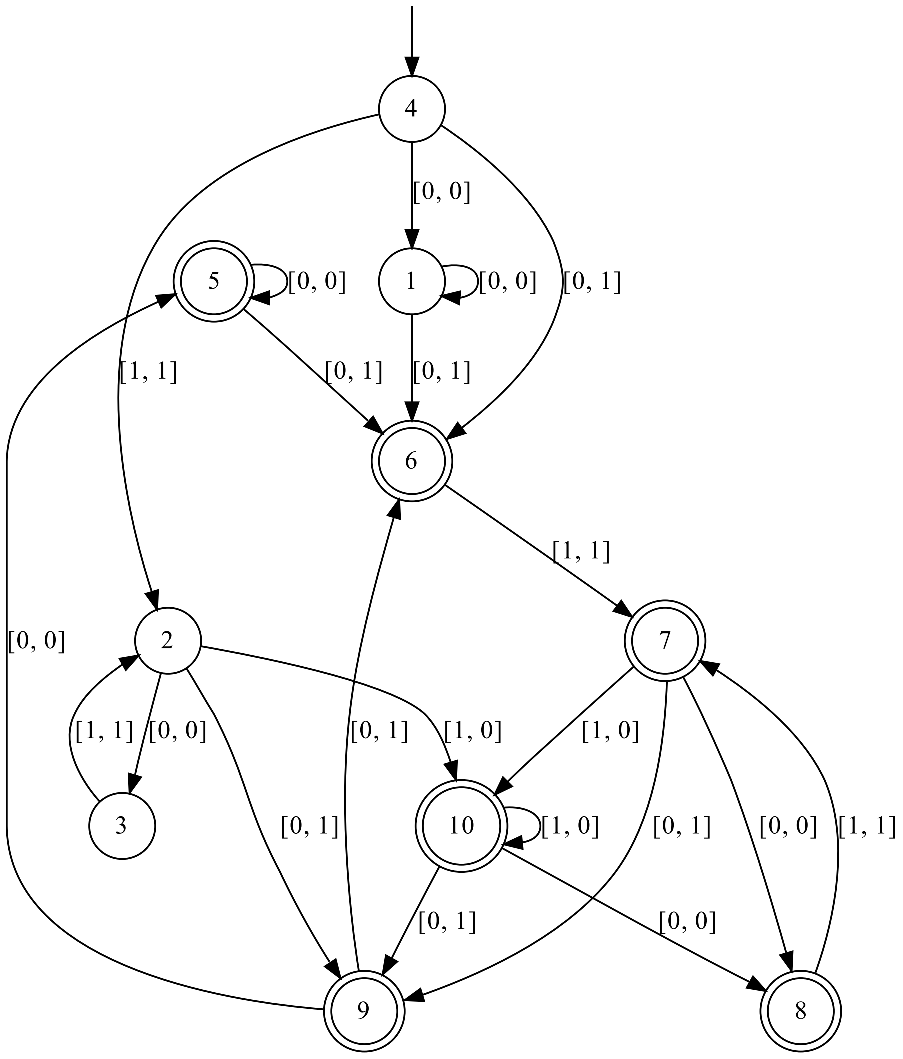

Figure 3.2: An automaton checking $(X \to Y) \wedge (X \neq Y)$ for ECA 30.

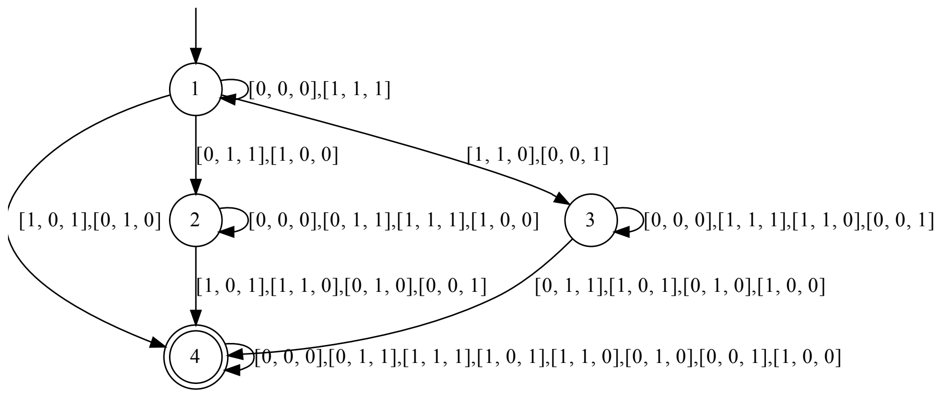

Figure 3.3: An automaton checking $(X \to Y) \vee (Y \to Z)$ for ECA 30.

most expensive operation in terms of running time and the size of the automaton generated.

Determining the validity of a sentence obviously requires a method of intepreting quantifiers and evaluating the resulting sentence. Existential quantifiers are handled by *projection*: erasing the track corresponding 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. Sentences with universal quantifiers can be verified by converting the universal quantifiers to existential quantifiers. Universal quantifiers are thus doubly expensive: converting the expression $\forall Y\ \exists X : X \to Y$ to the equivalent form $\neg\ \exists X : \neg(\exists Y : X \to Y)$ may require us to perform complementation twice. Once all bound variables have been projected, the resulting automaton can be checked for emptiness in linear time. An accepting (infinite) path is a witness for the existential quantifiers in the checked formula.

This completes the theoretical framework for verifying assertions about the behavior of one-dimensional cellular automata. The extensive use of exponential and super-exponential algorithms indicates that computational feasibility will be a critical question of any implementation.

Chapter 4

Implementation

The model-checking system is written primarily in C++, and consists of three separable components. The first is an independent library for manipulating Büchi automata, including routines for determinization and complementation via Safra's construction. The core of the model-checker constructs Büchi automata which correspond to logical sentences and manipulates them using that library. This is wrapped in a simple command-line interface which uses a parser (specified using Bison) to interpret formulas input by the user. Our intention is to make all code from this implementation available under the GNU General Public License.

4.1 Automata Library

Considering the significant complexity of Büchi complementation algorithms, an efficient procedure is extremely important. Of the two algorithms for complementation previously discussed, we chose to rely primarily on the determinization-based algorithm using Safra's construction so as to construct only the accessible portion of the complement automaton. We implemented Safra's construction as a standalone package, so that future work on ω-automata can easily reuse our implementation of this operation.

4.1.1 Improvements to Safra's Construction

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 $2^{O(n \log n)}$, any optimiza-

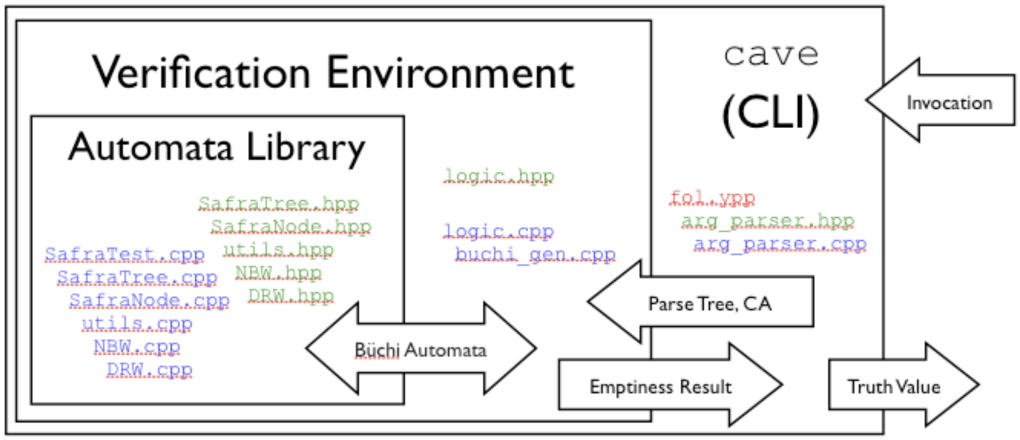

Figure 4.1: Overview of implementation details and flow of information.

tions which can be made are important. We present several such optimizations to Safra's construction below. These help to reduce the size of the Rabin automata or to speed up the construction procedure.

Transition Ordering

The order of the first two steps of Safra's construction may be exchanged without affecting the proof of correctness presented in [26]. 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 reverses the order of these steps from the literature implementation. 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.

Marking New Nodes

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. A marked node corresponds to 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 generates additional states in the Rabin automaton. Eventually the node will become marked in the final step of Safra's construction, but additional transitions are required to reach that point without this optimization.

Single Traversal Transition

Rather than implementing each of the steps of Safra's construction separately, we can perform them all with a single traversal of the Safra tree. Each step requires only knowledge about left siblings, children, and parents of the current node, so the entire procedure can be accomplished with a single in-order traversal. This significantly accelerates the transition process in Safra's construction. Since this may result in the creation and immediate removal of new child nodes, it is important to update a list of available node names to maintain consistency.

State Reduction Analysis

To characterize 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 2.2, 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 4.2 demonstrates the unoptimized result of Safra's construction that is presented in [1]. In Figure 4.3, we demonstrate the result of marking new nodes when performing the same determinization, and Figure 4.4 demonstrates the use of all our optimizations in the construction.

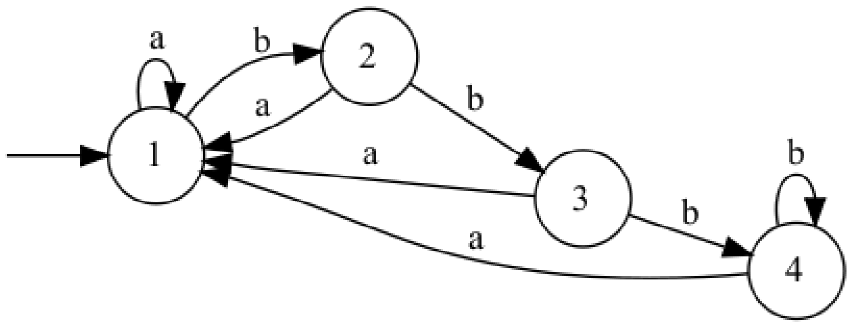

Figure 4.2: Rabin automaton equivalent to the Büchi automaton in Figure 2.2, produced by the unmodified version of Safra's construction. The acceptance condition for this automaton is $\{(\{1, 2\}, \{4\})\}$.

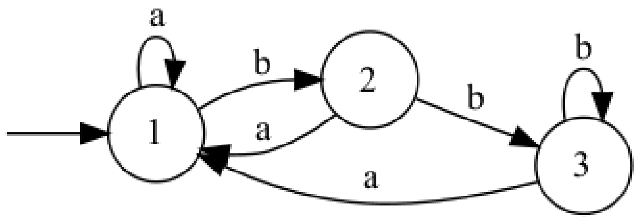

Figure 4.3: Result of marking new nodes when determinizing the automaton from Figure 2.2. The acceptance condition for this automaton is $\{(\{1, 2\}, \{3\})\}$.

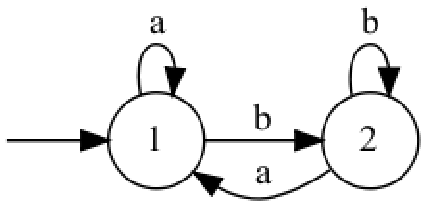

Figure 4.4: Optimal Rabin automaton equivalent to the automaton from Figure 2.2, produced using the optimizations to Safra's construction described in Section 4.1.1. The acceptance condition for this automaton is $\{(\{1\}, \{2\})\}$.

4.2 CAVE: Cellular Automata Verification Environment

The model-checking system as a whole is named CAVE, for the Cellular Automata Verification Environment. It recognizes sentences from a subset of first order logic, and parses them according to the following grammar:

$$\begin{array}{lcl}
literal & ::= & VARIABLE \rightarrow VARIABLE \\
 & | & VARIABLE = VARIABLE \\
 & | & \neg literal \\
quantifier & ::= & \forall\ VARIABLE \\
 & | & \exists\ VARIABLE \\
 & | & \neg quantifier \\
quantifier\_list & ::= & quantifier \\
 & | & quantifier\ quantifier\_list \\
conjunction & ::= & literal \\
 & | & literal \wedge conjunction \\
 & | & quantifier\_list\ conjunction \\
disjunct & ::= & conjunction \\
 & | & conjunction \vee disjunct \\
formula & ::= & disjunct \\
 & | & quantifier\_list\ (\ formula\ )
\end{array}$$

This grammar is dictated by the reality of constructing Büchi automata to model formulas. In particular, it is primarily designed to support formulas in prenex normal form with the matrix written in conjunctive normal form, a formula configuration which greatly simplifies automata construction. As can be seen in the grammar, though, there is a limited amount of support for other formulas. When the same property can be expressed in different ways, users should remember to take advantage of the asymmetrical cost of checking different formula structures.

Interaction with CAVE is accomplished through a minimal command-line interface. Currently the model-checking procedure is only enabled for elementary cellular automata, although a small amount of additional work will extend this to all cellular automata over binary alphabets. The algorithms we use can be extended to the bi-infinite case easily when the formulas being checked do not involve negation, so there is currently support for checking a very few formulas in the bi-infinite case as well. This should be regarded as experimental as its correctness has not been extensively tested.

Chapter 5

Experimental Results

The proof of decidability for model-checking omega cellular automata was previously presented in [34]. As we discussed earlier, the question of tractability was still open due to the exponential nature of some model-checking algorithms. In this chapter, we present experimental results suggesting that model-checking is feasible for many formulas in models defined using elementary, one-way infinite cellular automata.

5.1 Constructing Rabin Automata

Evaluating the efficiency of our implementation of Safra's construction is important due to its complexity and importance for complementation in the model-checking algorithm. However, attempting to find a good method for analyzing the effects of our improvements raises questions about the generations of random graphs and arbitrary automata. Suggestions for graph generation and a review of the difficulty of the problem can be found in [15] and [1].

In order to provide a general assessment of the efficiency of our improvements, we determinized some of the sample automata presented in [1]. Figure 5.1 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, while using our optimizations generates an automaton of 256 states. The transition matrices of the Rabin automata are shown in Figure 5.3.

Another example from the same source is an automaton which blows up to a total size of 13696 states under classical determinization. With our optimizations, this is reduced to 10777 states. The Büchi automaton is presented in Figure 5.2, the unoptimized determinization in Figure 5.4, and the optimized determinization in Figure 5.5.

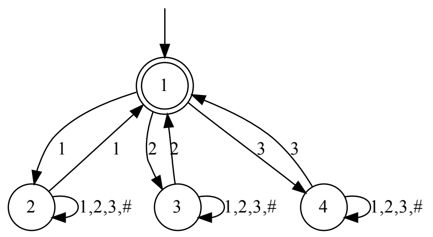

Figure 5.1: A Büchi automaton which exhibits an exponential blowup in state size under determinization. Determinization results are presented in Figure 5.3.

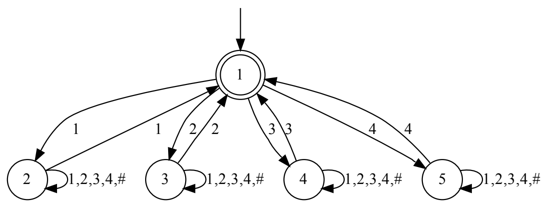

Figure 5.2: Another Büchi automaton exhibiting exponential blowup in state size under determinization. Determinization results are given in Figures 5.4 and 5.5.

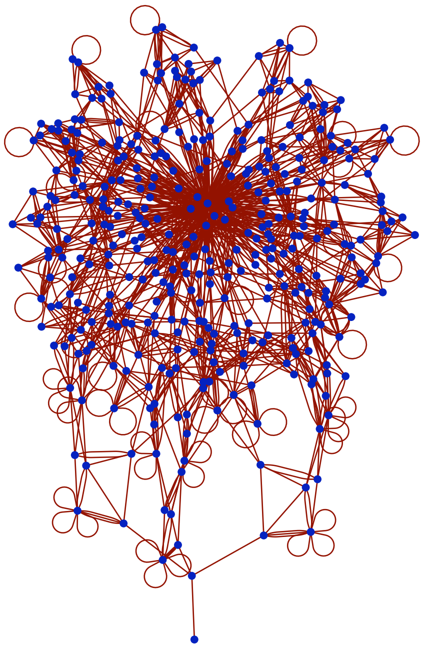
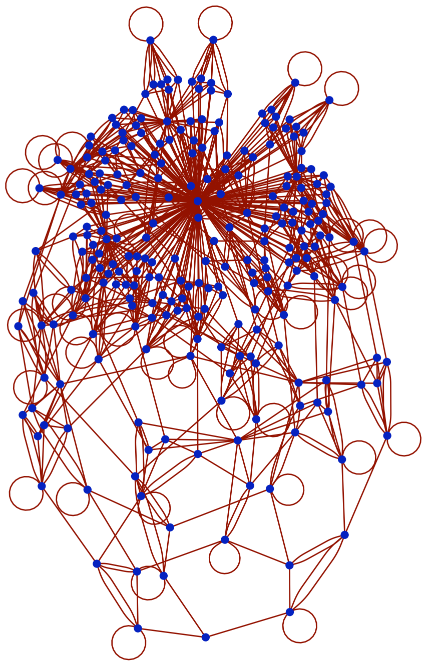

Figure 5.3: Left: The transition matrix of the Rabin automaton resulting from the determinization of the automaton in Figure 5.1 without optimization contain 384 states. Right: the transition matrix of the Rabin automaton resulting from the same determinization using our optimizations contains 256 states.

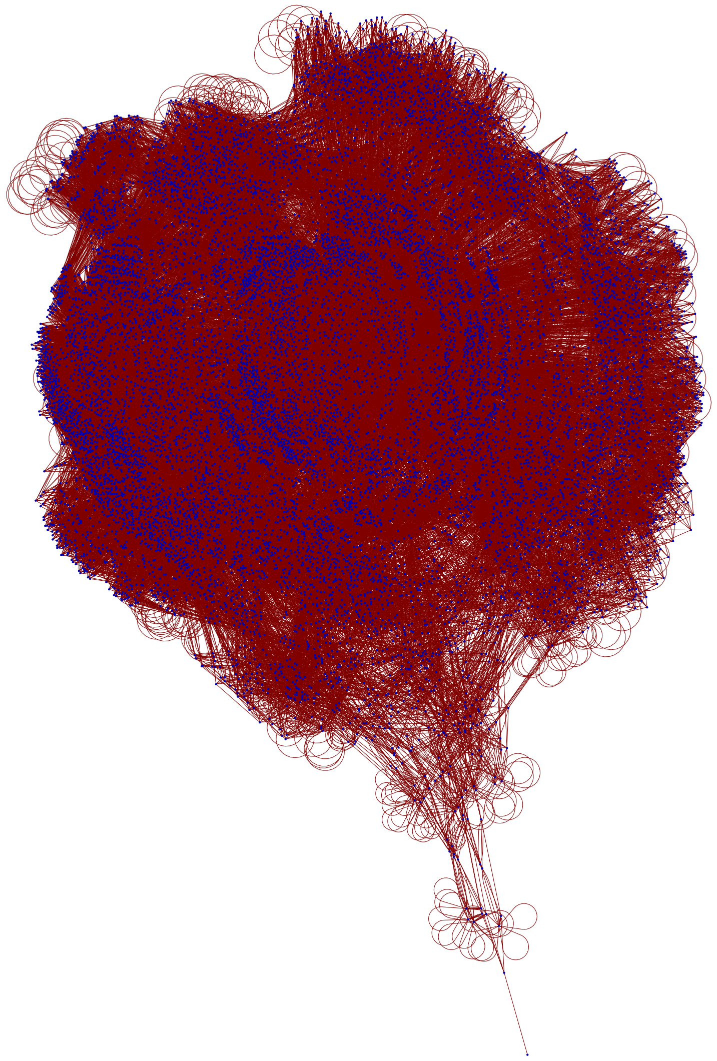

Figure 5.4: The transition matrix of the Rabin automaton resulting from the determinization of the automaton in Figure 5.2; it contains 13696 states. An interesting component is the structure to the bottom of the diagram, which includes the initial state and a sink state.

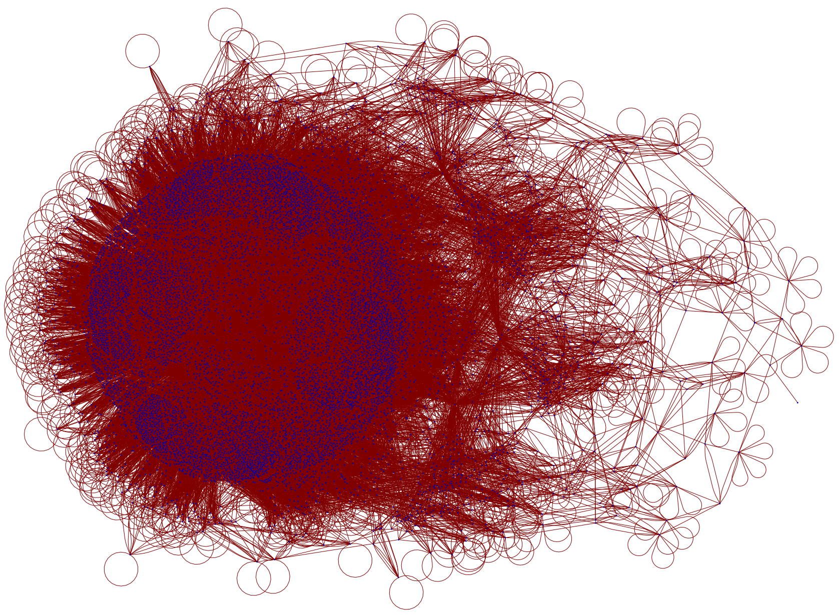

Figure 5.5: The transition matrix of the automaton resulting from the determinization of the automaton in Figure 5.2 if our optimizations are used. It contains 10777 states, which is an improvement of 21% over the unoptimized determinization.

5.2 Performance

The performance of our implementation in terms of time and memory is even more important than the size of the Rabin automata generated. In this section, we present profiling results for the model-checking algorithm. To characterize the performance on a set of formulas involving a variety of different operators, we used the formulas for injectivity, surjectivity, and “there exists a k-cycle”. The test machine for the profiling operations was equipped with two dual-core Intel Xeon processors at 3.00 GHz and 2 gigabytes of RAM. Our results indicate that many simple formulas can be checked in less than two minutes on this hardware.

The time required to check the existence of a k-cycle for all elementary cellular automata and values of k up to 7 is summarized in Figure 5.6. The graph was generated with a timeout of 120 seconds; only four formulas required more than this time to check and are not shown on the graph.

Memory usage was generally very low when checking these formulas. Figure 5.7 shows a memory profile over time for checking one of the more difficult formulas that we examined, the existence of a 6-cycle for ECA 90. The spikes in the center correspond to the points where the algorithm adds one of the inequality clauses (required to ensure that the 6-cycle is a true 6-cycle and not a series of 2-cycles, for example). Adding each such clause doubles the size of the automaton, and then causes the construction of the transitive closure of the transition graph to check for unreachable states. We selected ECA 90 for this demonstration purpose because the algorithm is not able to eliminate any states during this procedure, and thus this particular formula is one of the more difficult examples. If the algorithm were able to eliminate states, there would be a drop in memory usage after each spike rather than a return to the plateau.

In order to determine which portion of the model-checking algorithm consumes the most computation time, we profiled the CPU usage while checking injectivity of each of the elementary cellular automata. Our results, shown in Figure 5.8, show that just over half of the program’s time is spent constructing the transition automata. Just under half is spent performing complementation operations, with Safra’s construction consuming slightly more time than the conversion back to Büchi automata. It should be noted that these proportions are completely dependent on the formulas being checked, and that checking many properties, such as existence of a k-cycle, does not require any complementation.

Finally, we show that Safra’s construction does not utilize a prohibitively high amount of memory. Figure 5.9 shows the memory profile over time for constructing the Rabin automaton of 10777 states shown in Figure 5.5. We did not en-

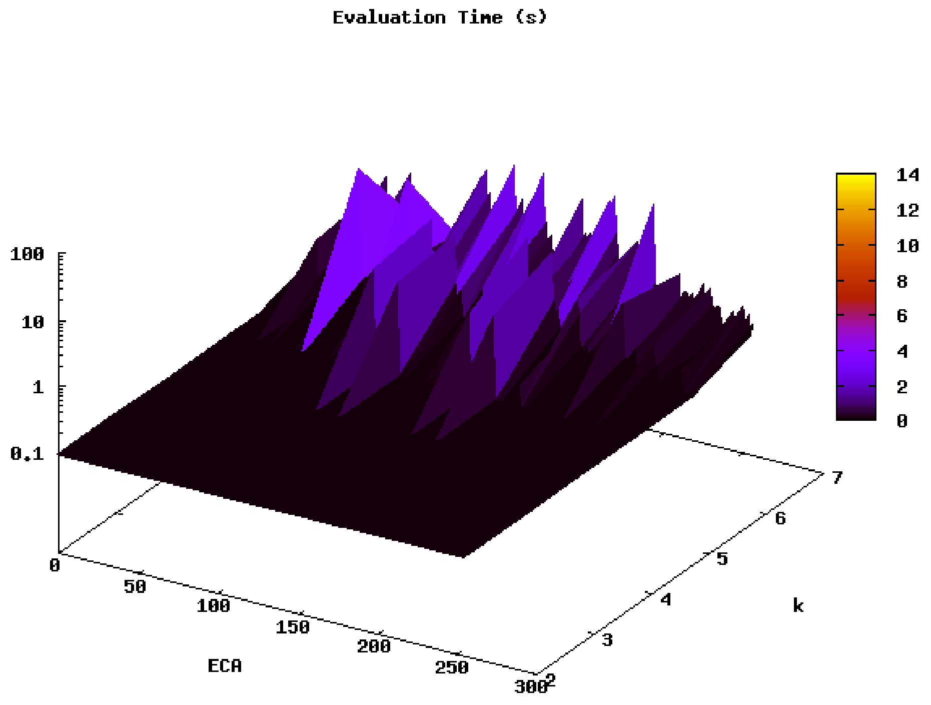

Figure 5.6: Time required to check the existence of a *k*-cycle.

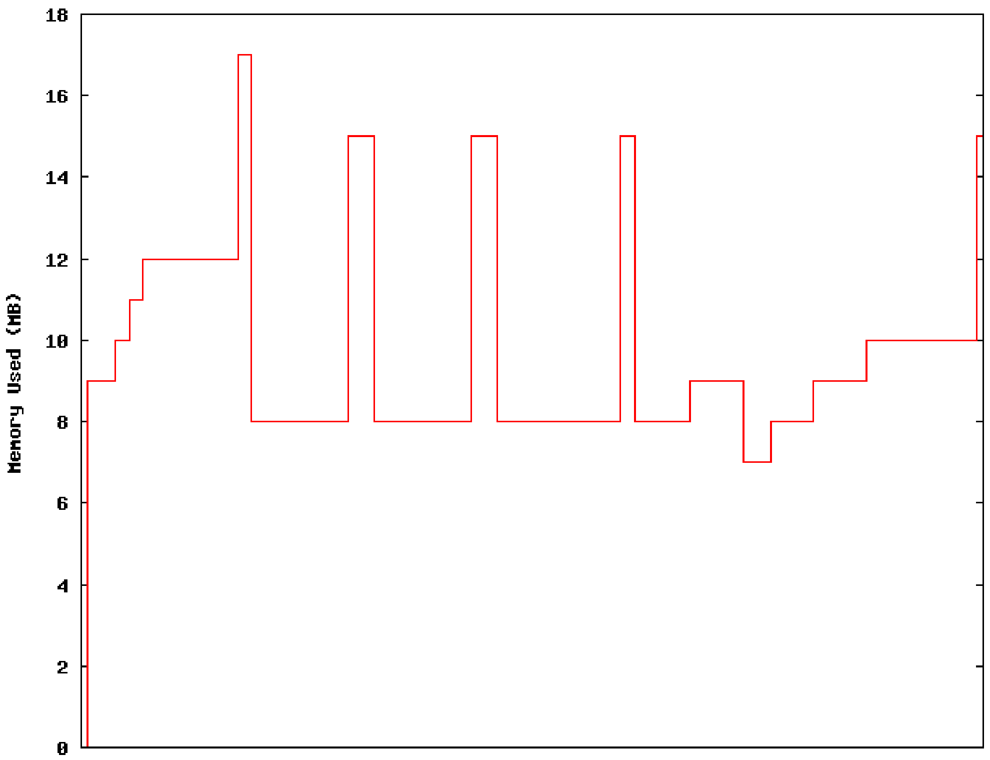

Figure 5.7: Memory usage over program execution when checking existence of a 6-cycle for ECA 90.

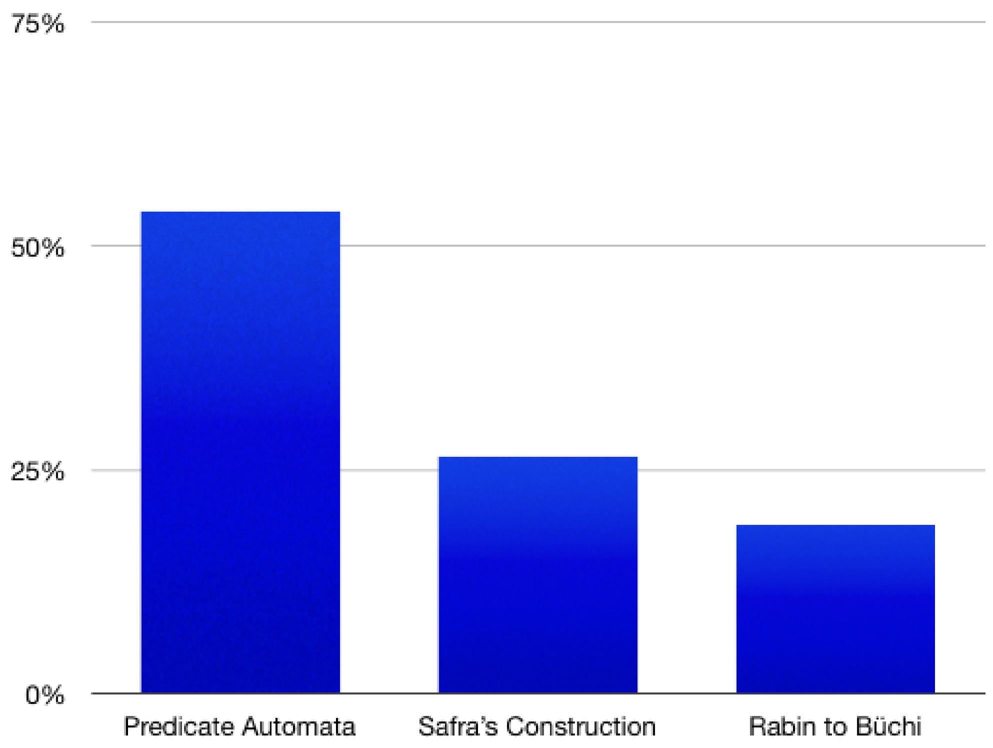

Figure 5.8: CPU profile of CAVE when checking injectivity of all elementary cellular automata.

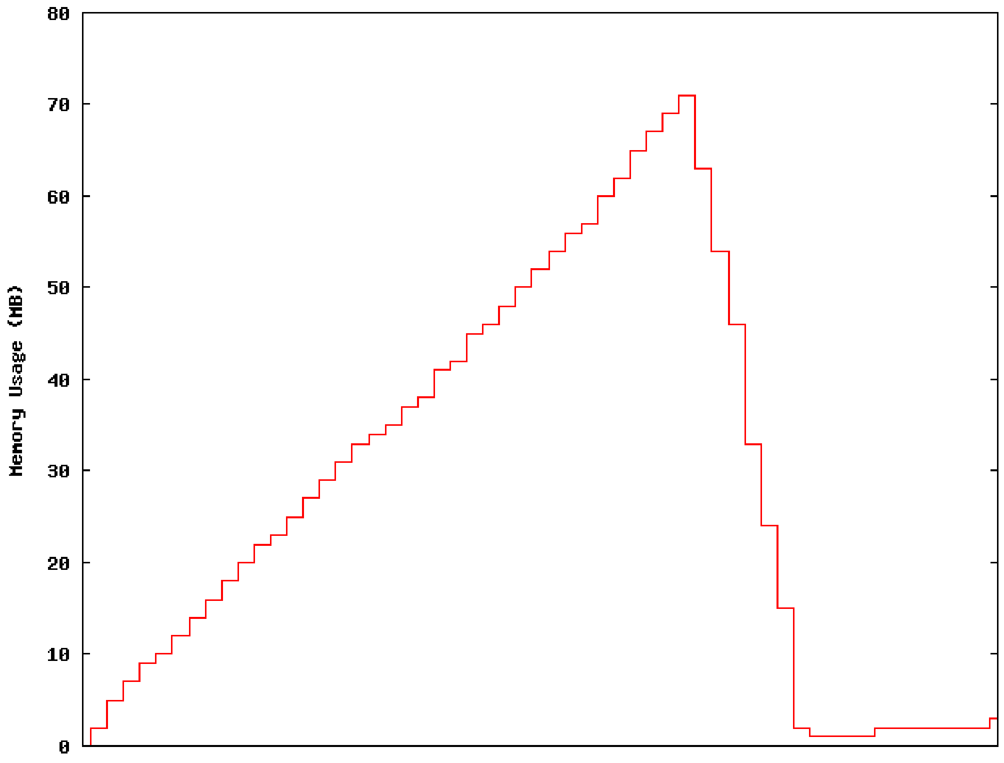

Figure 5.9: Memory usage over program execution when constructing the Rabin automaton shown in Figure 5.5.

counter any automata of this size in practice while checking our example formulas. However, the peak memory utilization of approximately 70 megabytes suggests that the memory requirements for Safra's construction are also not an insurmountable challenge. Additionally, the graph demonstrates that the vast majority of the memory used by Safra's construction can be freed as soon as the graph is fully explored, since it is used to store Safra trees and node information. Maintaining the Rabin automaton in memory requires significantly less storage.

Chapter 6

Conclusions

We have made several contributions towards the study of cellular automata and ω-automata. The largest of these is the successful implementation of an extensible, open-source framework for model-checking cellular automata. We have also used this framework to demonstrate that the basic problem of model-checking omega cellular automata is not just decidable but also tractable for surprisingly many small formulas.

The results we presented in Chapter 5 are not conclusive, and represent a starting point for further study. However, they suggest that the limiting factor in checking formulas will be the number of variables involved in the construction. The performance of Safra's construction is not a bottleneck in the formulas we tested, and it may be that few of the "monster" automata exhibiting superexponential blowup during determinization arise naturally during the model-checking procedure. It is possible that memory constraints will become a concern when complementing larger automata under Safra's construction, but this was not a significant issue for the formulas we tested. Our conclusion is that model-checking omega cellular automata is a technically feasible problem.

Our implementation of the model-checking system CAVE is intended to be a contribution to 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 discuss how it could be improved in several ways. With continuing work, this system should be capable of proving 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.

6.1 Further Work

A number of significant improvements are possible to our model-checking system, some of which might greatly improve its performance or capabilities. The first of these is a minor programming task: extension of the program to handle cellular automata of arbitrary alphabets and radii. This will introduce additional exponential factors into the size of the automata generated, but will also increase the number of automata which the system can model. Additional extensions will require larger amounts of practical and theoretical work; those presented below are roughly arranged in ascending order of theoretical difficulty.

6.1.1 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 in [3] 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 infinite words over $\{a, b, c\}$ with finitely many as, finitely many bs, or finitely 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.

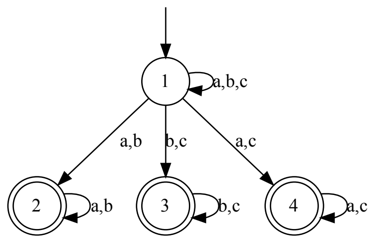

Figure 6.1: A Büchi automaton recognizing the language $(a + b + c)^*((a + b)^\omega + (b + c)^\omega + (a + c)^\omega)$.

6.1.2 Alternative Methods of 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 [28], is within $O(n^2)$ of the lower bound on complementation. Any work on Schewe's algorithm which allowed construction of only the accessible part of the complement automaton could be translated into a potential improvement in the efficiency of our model-checking algorithm. Work to avoid explicit determinization via the use of antichains, presented in [9] and [6], 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 [11]. Implementing the algorithm developed by Emerson and Jutla in place of Safra's construction has the potential to improve our constructions by an significant 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

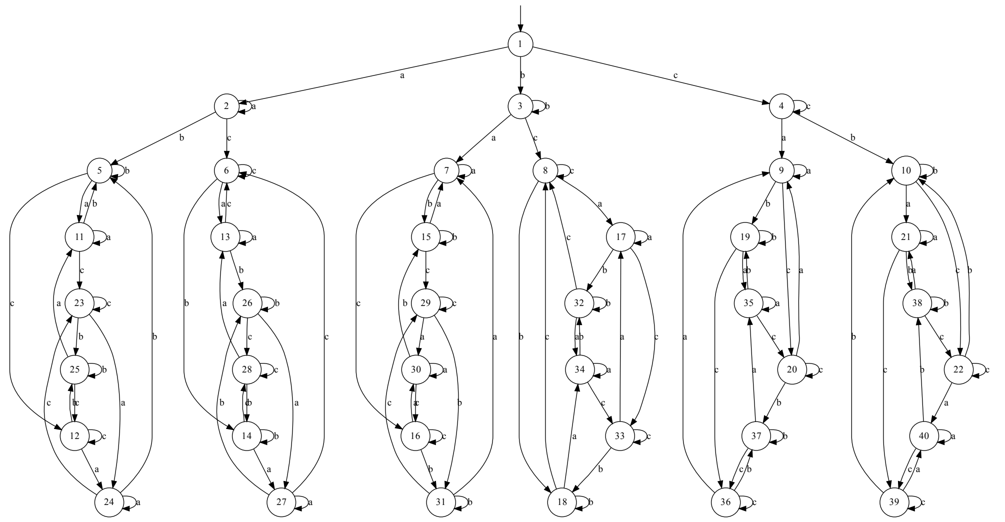

Figure 6.2: A Rabin automaton equivalent to the machine in Figure 6.1. Note the large number of strongly connected components, implying a high potential for parallelization.

and Jutla also extends to automata seen in practice.

6.1.3 ζ-Automata

The algorithm we use to check the evolution of one-way infinite cellular automata is extensible to the analysis of two-way infinite automata as well. In order to successfully transition from the infinite to the bi-infinite case, the model-checking system must be extended to complement bi-infinite automata. An algorithm for doing so is outlined in [14], but will require a significant amount of work to translate into practice. It also generates a large number of ω-automata to recognize the complement of a given ζ-language, introduces another significant performance factor to the model-checking procedure.

6.1.4 More Expressive Logics

In [34], 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 in the general case (see [30]).

An extension to this theory is presented in [12] 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.

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] S. Amoroso and Y. N. Patt. Decision procedures for surjectivity and injectivity of parallel maps for tesselation structures. *Journal of Computer and Systems Sciences*, (6):448–464, 1972.

[3] 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.

[4] J. R. Büchi. Weak second-order arithemtic and finite automata. *Z. Math. Logik and Grundl. Math.*, 1960.

[5] J. R. Büchi. On a decision method in restricted second-order arithemtic. *Logic, Methodology and Philosophy of Science*, pages 1–11, 1962.

[6] 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.

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

[8] K. Culik. Global cellular automata. *Complex Systems*, 9:251–266., 1995.

[9] 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.

[10] L. Doyen and J.-F. Raskin. Improved algorithms for the automata based approach to model-checking. In *Proceedings of TACAS 2007: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 4424, pages 451–465. Springer-Verlag, 2007.

[11] 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.

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

[13] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge UP., 2000.

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

[15] S. Janson, T. Łuczak, and A. Ruciński. *Random graphs*. Citeseer, 2000.

[16] J. Kari. Reversibility of 2D cellular automata is undecidable. *Physica D*, 45(397–385), 1990.

[17] B. Khoussainov and A. Nerode. Automatic presentations of structures. In *Logic and computational complexity*, pages 367–392. Springer.

[18] Joachim Klein and Christel Baier. Experiments with deterministic omega-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182 – 195, 2006. Implementation and Application of Automata, 10th International Conference on Implementation and Application of Automata (CIAA 2005).

[19] O. Kupferman. Avoiding determinization. *Proc. 21st IEEE Symp. on Logic in Computer Science*, 2006.

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

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

[22] 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.

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

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

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

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

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

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

[29] Klaus Sutner. A note on Culik-Yu classes. *Complex Systems*, 3(1):107–115, 1989.

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

[31] Klaus Sutner. De Bruijn graphs and linear cellular automata. *Complex Systems*, 5(1):19–30, 1991.

[32] Klaus Sutner. The size of power automata. *SLNCS*, 2136:666–677, 2001.

[33] Klaus Sutner. Cellular automata and intermediate reachability problems. *Fundam. Inf.*, 52(1-3):249–256, 2002.

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

[35] 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.

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

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

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

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