Model Checking Cellular Automata

Joe Gershenson

April 2, 2009

1 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.

2 Introduction

Cellular automata are valuable systems for modeling computations 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 procedure we use is presented by Sutner [11] as a constructive proof that model-checking cellular automata is decidable. It specifies properties of cellular automata as formulae in the first-order theory of phase-space, and provides a method for evaluating these properties. The natural predicate in this theory is the global map of the cellular automaton. To check predicates in the 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 this procedure uses Mathematica for high-level specification and interface purposes. The low-level construction 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 [9]. 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 machines.

3 Definitions and Background

Before presenting the details of our implementation, we briefly review the definitions of the automata involved in our constructions to avoid ambiguity.

3.1 ω-Automata

A cellular automaton is fully defined by its *local map* $\rho : \Sigma^{2r+1} \rightarrow \Sigma$ where r is the radius of the automaton and Σ the alphabet. By extending the local map to the entire set of cells, we can define a *global map* $G_\rho : \Sigma^{\mathbb{Z}} \rightarrow \Sigma^{\mathbb{Z}}$ (or $G_\rho : \Sigma^{\mathbb{N}} \rightarrow \Sigma^{\mathbb{N}}$, as appropriate), where the local map is applied to every cell. By a *configuration*, we will refer to a bi-infinite word over Σ, so we say that two configurations A and B are related by $A \rightarrow B$ if $G_\rho(A) = B$.

A Büchi automaton is the simplest extension of the theory of finite automata to one-way infinite inputs. Like a nondeterministic automaton on finite strings, a Büchi automaton is a tuple $(Q, \Sigma, \delta, I, F)$ where Q is the state set, Σ the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I \subseteq Q$ the set of initial states, and $F \subseteq Q$ the set of "final" states. Unlike an automaton on finite strings, however, a Büchi automaton accepts some word $w \in \Sigma^\omega$ if there is a run on w which reaches states in F infinitely often.

We refer to the set of words accepted by an automaton A as the language of A. Similarly, 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. 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 is provided in [7]) This shortcoming is especially critical since complementing a nondeterministic automaton requires determinizing it first.

A Rabin automaton is defined in order to generate a deterministic ω-automaton which can accept any recognizable language. A Rabin automaton is a tuple $(Q, \Sigma, \delta, i, R)$, where the state set Q, alphabet Σ, and transition relation δ are defined as for Büchi automata. The automaton is deterministic by definition, so δ defines at most one transition (q_1, σ, q_2) for each (q_1, 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, and is often referred to as a set of Rabin pairs. A run r is successful if there exists some index j such that r reaches E_j infinitely often and visits F_j only finitely often.

3.2 Deterministic ω-Automata

The equivalence of Büchi and Rabin machines for recognizing infinite words is given by McNaughton's Theorem:

Theorem 3.1. *Every 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 an arbitrary run returns to a final state in order to ensure that a singular run in the Büchi automaton reaches an accepting state infinitely often. A formal proof of correctness is given in [9], but we present the algorithm here for reference. Given a Büchi automaton $(Q, \Sigma, \delta, I, F)$, we build a Rabin automaton R where the states are labeled trees with marks on some nodes. Each node v in a tree is labeled with a nonempty subset of Q, denoted by $L(v)$, and 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 tree is one unmarked node labeled with I; if $I \subset F$, the tree is one marked node labeled I; otherwise, the tree is 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 name it with 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 result is a tree, which represents a new state of R. The total number of trees is bounded by $2^{O(n \log n)}$, where n is the size of the Rabin automaton, so we can explore the graph described by the state set and transition function to build the entire automaton in $2^{O(n \log n)}$ time. The acceptance condition (the set of Rabin pairs) defined as $\{(E_i, F_i) | i \in \mathit{nodes}\}$, where the state corresponding to tree T is in E_i if i is not a node present in T, and the state corresponding to tree T is in F_i if i is a marked node in T.

3.3 ζ - Automata

A ζ-Büchi automaton is a generalization of a Büchi automaton to accept two-way infinite word. As in [11] and [4], we define a ζ-Büchi automaton with a tuple $(Q, \Sigma, \delta, I, F)$ where the state set Q, alphabet Σ, and transition function δ are defined as for ordinary Büchi automata. A run on word $w \in \Sigma^{\mathbb{Z}}$ of a bi-infinite sequence of states $q \in Q$ can be indexed by $i \in \mathbb{Z}$, where $\delta(q_i, w_i, q_{i+1})$

for all i. The run is accepting if, for all i, there exists some $i' < i$ such that $q_{i'} \in I$ and there exists some $i'' > i$ such that $q_{i''} \in F$.

The ζ-Büchi automaton can recognize words from an $\omega\omega$-regular set: a set which can be written as a finite union of languages of the form ${}^{\omega}ABC^{\omega}$, where A,B,C are all regular languages. An $\omega\omega$-regular set is closed under complementation; the proof, found in [4], is again constructive and requires taking the union of a finite but exponential number of ω-automata. These ω-automata intuitively correspond to the languages ${}^{\omega}A$ and C^{ω}, and are complemented using Safra's construction to help obtain the complement of the $\omega\omega$ language.

4 Theory of Model Checking

The inspiration for this work is primarily due to the following theorem by Sutner:

Theorem 4.1. *Model checking for one-dimensional cellular automata is decidable.*

The proof, described in detail in [11], is again constructive. We construct a first-order structure for our logical theory with the space of configurations $\Sigma^{\mathbb{Z}}$ and the binary predicate $\rightarrow$. In order to express propositions, we use ζ-automata on bi-infinite words over alphabets of the form Σ^k. These machines distinguish words consisting of k tracks, where each track consists of a bi-infinite word over Σ that corresponds to a configuration state of the cellular automaton. The i^{th} character in the alphabet of the ζ-automaton, then, is a k-tuple containing the i^{th} character from each track.

For example, the machine for testing the atomic predicate $\rightarrow_\rho (A, B)$ scans words of the form $\{..., (A_{-1}, B_{-1}), (A_0, B_0), (A_1, B_1), ...\}$. It has state set $Q = (\Sigma^k)^{2r} \times (\Sigma^k)^r$ and a transition function where

$$((x_1, ..., x_{2r}, a), (y_1, ..., y_r, b)) \in \delta((x_0, ..., x_{2r-1}, y_0, ..., y_{r-1}), (a, b))$$

if $\rho(x_0, ..., x_{2r-1}) = y_1$. This corresponds to the natural mechanism of scanning the two tracks and ensuring that the second is derived from the first using local rule ρ. We note that all states in this machine are both initial and final, and that acceptance is precisely equivalent to the existence of a bi-infinite path.

Given two machines representing logical propositions ϕ and ψ, we can construct a new automaton to represent $(\phi \vee \psi)$ by taking the disjoint sum of the first two. Similarly, the product of two machines represents the conjunction $(\phi \wedge \psi)$ of the corresponding formulae. Existential quantifiers are handled by erasing the track corresponding to the variable being bound and checking the emptiness of the resulting automaton, and universal quantifiers are converted to existential quantifiers. Complementation is simply complementation of the corresponding ζ-automaton and is handled as described in the definition of that operation. Because of the exponential and super-exponential constructions involved in this, the complementation of propositions is by far the most expensive step in the model checking procedure.

5 Implementation of Safra's Construction

We implemented Safra's construction as a standalone Java package, so that future work on ω-automata can easily reuse an efficient implementation of this operation. The running time and memory usage of the determinization algorithm are obviously related to the size of the generated Rabin automaton. As a relative benchmark, Rabin automata with approximately one million states require approximately three minutes to generate on a 2.4 GHz Intel Core 2 Duo processor. Since a Rabin automaton of that size can be generated from the determinization of a Büchi automaton with only five states, it is clear that any optimizations which can be made are critically important.

5.1 Improvements

Our implementation of Safra's construction presents several improvements over the original definition or previous implementations in the literature [1, 9]. These improvements prove to be necessary in practice for dealing with large machines.

First, we implicitly define a sink state. No longer requiring the automaton to be complete allows a massive reduction in the number of transitions stored in memory, as well as simplifying further operations such as product constructions.

Second, the order of steps 1 and 2 of Safra's construction may be exchanged without affecting the proof of correctness. While our definition of the procedure for transitioning tree T calls for transitioning the labels of each node before creating new children, previous implementations have constructed new children of each node first and then transitioned each label. By reversing the order of those steps, we maintain the correctness of Safra's algorithm but reduce the size of the typically resulting automata.

Third, nodes which are created in step 2 should be marked. This is also compatible with the proof of correctness of the algorithm. Informally, a marked node represents a 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. If a new node n is not marked, we may have to perform additional transitions (generating and storing additional trees) until a sufficient number of additional children of n are added such that n becomes marked in step 5.

Finally, an advanced heuristic for node renaming can be employed to further reduce the size of the resulting Rabin automata. Whenever 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, but this is asymptotically insignificant compared to the benefit of reducing the size of the automaton.

5.2 Results

To illustrate these simple improvements, we demonstrate the determinization of the Büchi automaton in Figure 1, which accepts all words over $\{a, b\}$ that contain only finitely many occurrences of b. This is a canonical example which is also used to demonstrate why power set construction fails for Büchi automata.

The determinization of this automaton using the improvements described above generates the Rabin automaton in Figure 2. Determinization of the automaton without those improvements produces the Rabin automaton in Figure 3.

The improvement in computational performance from using these optimizations is significant on larger automata. The unimproved version of Safra's construction produces a Rabin automaton of 13696 states for the Büchi automaton in Figure 4. Our implementation produces an equivalent automaton of 10776 states, representing a compression of approximately 21% over previous methods. In this extended abstract, we ask the reader to ac-

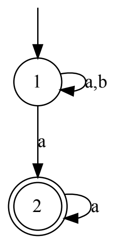

Figure 1: Büchi automaton for accepting finitely many b's.

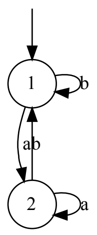

Figure 2: Smallest Rabin automaton corresponding to the Büchi automaton in Figure 1. The only Rabin pair in the acceptance condition is $(\{1\},\{2\})$

cept our assurances that this is a typical example. In the full thesis we will present detailed statistics on the improved performance of the algorithm on various types and sizes of automata.

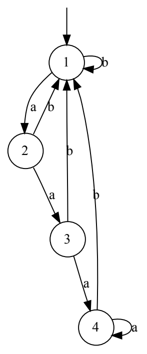

Figure 3: A larger Rabin automaton corresponding to the Büchi automaton in Figure 1. The only Rabin pair in the acceptance condition is $(\{1,2\},\{4\})$.

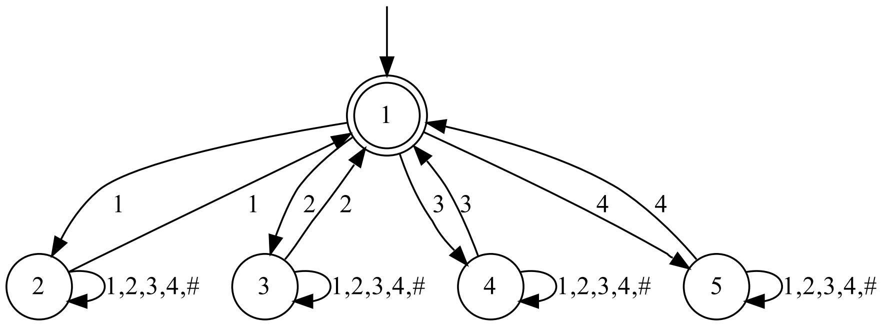

Figure 4: This Büchi automaton, presented in [1], is known to be equivalent to a Rabin automaton of 13696 states. We demonstrate how to construct an equivalent automaton of 10776 states, representing a 21% improvement.

6 Implementation of Model Checking

The model checking engine is also 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 backend written in Java.

As an example of the use of the model checking system, we demonstrate how to check that the global map of a given automaton ρ is surjective. In the theory of phase-space, this property can be expressed as

$$\forall x \; \exists y : y \to_\rho x$$

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

$$\neg\exists x \; \neg\exists y : y \to_\rho x$$

parsing the components of the formula appropriately,

$$\neg(\exists x(\neg\exists y(y \to_\rho x))))$$

and converting the resulting formula into a series of operations to pass via JLink to the model checker.

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

Obviously, the preamble `setCA` allows 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 path 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 [6, 10]), so generating such configurations automatically is one possible use of our model checking application.

7 Extensions

While we have designed and implemented a powerful system for the study of cellular automata, several natural extensions to this research are apparent. First, there are additional algorithms for performing operations on ω-automata which might improve performance. In particular, a method for simultaneous determinization and complementation, proposed in [3], might have the potential to improve the efficiency of our model checking algorithms.

Along similar lines, in certain cases it may be possible to check universality much more efficiently through the use of antichains [2]. If we could check universality efficiently, it would remove the need to replace universal quantifiers with existential quantifiers and, more importantly, decrease the number of additional complementation operations.

Finally, and most ambitiously, we know that the reachability relation, "configuration x eventually evolves to configuration y" is undecidable in the general case. However, it may well be decidable for simple cellular automata, such as the linear elementary cellular automata represented by rules 90 and 150. The theoretical machinery required to answer such a question or characterize precisely which cellular automata might be amenable to such treatment would be a significant contribution.

References

[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] 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.

[3] E.A. Emerson and C.S. Jutla. On simultaneously determinizing and complementing omega-automata. pages 333–342, Jun 1989.

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

[5] 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).

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

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

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

[9] S. Safra. On the complexity of omega-automata. pages 319–327, Oct 1988.

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

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