RICE UNIVERSITY

Experimental Evaluation of Explicit and Symbolic Automata-Theoretic Algorithms

by

Deian Tabakov

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

Approved, Thesis Committee:

Professor Moshe Y. Vardi, Chair
Karen Ostrum George Professor
Department of Computer Science

Professor Devika Subramanian
Department of Computer Science

Assistant Professor Luay K. Nakhleh
Department of Computer Science

Houston, Texas

December 2005

Abstract

Experimental Evaluation of Explicit and Symbolic Automata-Theoretic Algorithms

by

Deian Tabakov

The automata-theoretic approach to the problem of program verification requires efficient minimization and complementation of nondeterministic finite automata. This work presents a direct empirical comparison of well-known automata minimization algorithms, and also of a symbolic and an explicit approach to complementing automata. I propose a probabilistic framework for testing the performance of automata-theoretic algorithms, and use it to compare empirically Brzozowski's and Hopcroft's minimization algorithms. While Hopcroft's algorithm has better overall performance, the experimental results show that Brzozowski's algorithm performs better for "high-density" automata. In this work I also analyze complementation by considering automaton universality as a model-checking problem. A novel encoding presented here allows this problem to be solved symbolically via a model-checker. I compare the performance of this approach to that of the standard explicit algorithm which is based on the subset construction, and show that the explicit approach unexpectedly performs an order of magnitude better.

Acknowledgments

I would like to extend my sincere gratitude to my advisor Professor Moshe Y. Vardi. Dr. Vardi is one of those people who can encourage you and motivate you with a single word, a gesture or even a single glance. Without his guidance and support this work would not have existed. I would like to thank Dr. Devika Subramanian and Dr. Luay K. Nakhleh for being members on my thesis committee. I am also thankful to Dr. Kousha Etessami and to the Laboratory for Foundations of Computer Science where I spent one summer as a visiting student.

I owe a lot to Guoqiang Pan and Dr. Doron Bustan who have helped me times and again understand a difficult theorem, algorithm or concept. Special thanks to all past and present students in the Verification and Algorithms groups who helped me find my bearings in the department and made me feel at home. Last but not least, I would like to thank my parents, Evelina and Todor, for their support and encouragement.

This work was supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, and ANI-0216467, by BSF grant 9800096, by Texas ATP grant 003604-0058-2003, and by a grant from the Intel Corporation.

Contents

List of Figures

Chapter 1

Introduction

On November 2, 1988, a first-year Cornell graduate student named Robert Morris unleashed a computer program that crippled the Internet and caused damages worth millions of dollars in lost productivity. Morris' program exploited a bug in the *fingerd* daemon which allowed it to mount a buffer-overflow attack. The program queried *finger* with a string that was carefully chosen to overwrite the buffer allocated for input and to modify the stack frame. Instead of returning normally, *fingerd* was routed to a procedure within the invading string. The new procedure executed `/bin/sh`, which gave the attacker a remote shell on the machine, from which the attack was repeated.

While Morris' program was the first well-known exploit of a buffer-overflow error, it was certainly not the last. The sad experience from the last twenty years is that software (and hardware) bugs can be both very expensive and extremely difficult to find. The usual testing techniques involve running the program (or a model of the hardware) on carefully chosen inputs. The problems with this approach are threefold. First, as designs get more complicated, coming up with tests that provide good coverage becomes extremely hard. Second, the number of required tests tends to grow very quickly as the design evolves, thus driving up the cost of testing. Finally, testing can give us only *confidence* in the implementation, but cannot *prove* that all bugs have been found. Thus, there has been a tremendous push for efficient algorithms and techniques that allow one to prove that a program satisfies certain properties. The process of stating and proving properties about programs is known as *program verification* and it is the broad field within which this work is developed.

The two main types of program verification are *proof-based* and *model-based* verification. This work follows the latter approach, so I give only a brief overview of the proof-based approach. The proof-based approach is used primarily for programs that are expected to terminate and produce a result. We start with a set of known mathematical axioms and facts that have already been proven in some proof system and then try to produce a derivation that leads to the property. While the bulk of the work can be automated, it still requires an experienced user to help guide the proof. A well-known tool that exemplifies the proof-based approach is the ACL2 theorem prover designed by Moore [KMM00a, KMM00b]. One of the early success stories of this theorem prover dates back to 1998, when Moore, Lynch and Kaufmann successfully used ACL2 to verify the floating-point division instructions of the AMD-K7 processor [MLK98]. ACL2 has since been used to verify the register-transfer level specification of AMD-K7's multiplication and square-root instructions [Rus98], and properties of Java code [Moo03].

One of the advantages of the model-based approach to verification is that it is more amenable to automatic execution. Given a program P and a property φ, we check whether the program satisfies (is a model of) the property [CGP99]. This high-level description does not specify how the check should be done and indeed the last twenty years have witnessed the development of multiple techniques. One of the leading techniques has been the *automata-theoretic approach*, originally proposed by Vardi and Wolper [VW86]. Intuitively, Vardi and Wolper suggest that we view the program P as a finite-state generator of words, and the specification φ as a finite-state acceptor. Then the model-checking problem is reduced to an automata-theoretic question: whether the automaton $A_P \cap A_{\overline{\varphi}}$ empty [VW86]. My thesis follows this line of research by investigating the complementation of the property automaton $A_{\overline{\varphi}}$, that is, deriving $A_{\overline{\varphi}}$ from A_{φ}.

1.1 Approach of the Thesis

In this section I present briefly the contributions of the thesis. First I propose a general-purpose probabilistic framework for evaluating automata-theoretic algorithms and use it for the analysis of the algorithms in the subsequent chapters. Second, I analyze the complementation step of the automata-theoretic approach (deriving $A_{\overline{\varphi}}$ from A_{φ}) by reducing the problem to universality checking, and compare the classical approach to a novel approach for solving this problem. Finally, I consider the special case when φ is a *monitor* for a *safety property.* I compare two classical algorithms for automata canonization, which allow us to construct efficient monitors from the same property that we use for model checking.

1.1.1 Probabilistic Framework for Testing Automata-Theoretic Algorithms

For many algorithms the asymptotic complexity does not tell the whole story because it hides constant factors. Especially in the case when two algorithms have the same asymptotic complexity, it is important to know how they compare in practice. Making this comparison for automata-theoretic algorithms is complicated by the fact that there are no good benchmarks and it is not even clear what types of automata should be included in such benchmarks.

In this work I propose evaluating automata-theoretic algorithms based on their performance on randomly generated *non-deterministic finite automata* (NFA). This contribution is inspired by recent work on randomly generated problem instances [CKT91], for example, random 3-SAT [SML96]. The hardness of the instances can be varied by controlling their density. In the case of NFA, there are two densities to control: the density of the accepting states (i.e., ratio of accepting states to total states) and the density of transitions (i.e., density of transitions per input letter to total states). For both densities I propose using constant ratios, which yield linear densities. For simplicity, this work assumes a unique initial state.

In addition to automata with uniform distribution of transitions, I also consider automata with locality. I introduce two models of locality–*linear* and *matrix structured automata*. These models restrict the transitions to a neighborhood of states, thereby providing a “structure” for the automaton. The linear and the matrix models are designed to allow a more comprehensive comparison of symbolic automata-theoretic algorithms.

It is not a priori clear that the linear-density model is an interesting model for studying automata-theoretic algorithms. I show empirically that this probability model does yield an interesting problem space. On one hand, the probability of universality does increase from 0 to 1 with both acceptance density and transition density. (Unlike the situation with random 3-SAT, the probability here changes in a smooth way and no sharp transition is observed.) On the other hand, the size of the canonical deterministic finite automata (DFA) does exhibit a (coarse) phase transition with respect to the transition density of the initial NFA, peaking at density 1.25. (It is interesting to note that random directed graphs with linear density are known to have a sharp phase transition with respect to connectivity at density 1.00 [Kar90].) The scaling of the size of the canonical DFA depends on the transition density, showing polynomial behavior for high densities, but super-polynomial but subexponential behavior for low densities.

1.1.2 Universality Checking as Model Checking

Suppose that the property φ is given as a non-deterministic finite automaton A_φ. In this case most of the time spent constructing $A_P \cap A_{\overline{\varphi}}$ is dedicated to constructing $A_{\overline{\varphi}}$. In order to analyze this step of the construction, I consider a simplified setting: let P, when viewed as a generator of words, be the *universal automaton*. In this instance the model-checking problem is reduced to checking whether A_φ is also universal. This is know as the "universality problem" and has been proved to be PSPACE-complete [MS72].

This work uses two approaches, explicit and symbolic, for solving the universality problem. The standard way to check for universality is to determinize the automaton explicitly using the subset construction and then to check if a rejecting set is reachable from the initial state. This is referred to as the *explicit* approach. This work studies the explicit approach by using the Java tool `Automaton.brics.dk` [Mø04] and the model-checker `SPIN` [Hol97, Hol04]. When evaluating the explicit approach, I present data for both `SPIN` and `Automaton`.

In addition to the explicit approach, this work studies a symbolic approach to the universality problem. I introduce a novel method, which reduces the universality problem to model checking of a *safety property* (defined in Chapter 4), enabling us to apply *symbolic* model checking algorithms [BCM+92]. These algorithms use Binary Decision Diagrams (BDDs) [Bry86], which offer compact encoding of Boolean functions. To solve universality symbolically, I view the determinized automaton as a synchronous sequential circuit. The reachability-of-a-rejecting-set condition can be expressed as a temporal property of this digital system. Thus, the universality problem can be reduced to a model-checking problem and solved by a symbolic model checker; I used two versions of `SMV`–`Cadence SMV` [Cad] and `NuSMV` [CCG+02]. To get the best possible performance from the model checker, several optimization techniques were considered, including the encoding of the determinized automata as a digital system, the representation of the transition relation, the order of traversal of the model, and the order of the BDD variables. The experiments presented here used the configuration that led to the best performance of each tool.

The conventional wisdom in the field of model checking is that symbolic algorithms typically outperform explicit algorithms on synchronous systems, while the latter outperform on asynchronous systems. In view of that, I expected the optimized symbolic approach to outperforms the explicit, rather straightforward approach. Surprisingly, the empirical results show that the conventional wisdom does not apply to the universality problem, as the explicit algorithm dramatically outperformed the symbolic one.

1.1.3 Monitor Construction

While model-checking is essentially a proof that the program satisfies the property, such confidence does come at a price. Sometimes one does not have the resources to perform a complete model-checking and has to resort to running a simulation of the model on test cases. In some instances it is possible to reuse a specification of the program during the testing phase. This work focuses on one special type of properties– those asserting that something "bad" does not happens during the execution of the program. Such properties are known as *safety properties.* For example, the property "the length of every string is smaller than the size of the allocated buffer" is a safety property. This property is violated in programs with a buffer-overrun bug like the version of `fingerd` that Robert Morris exploited in 1988.

Intuitively, φ is a safety property if every violation of φ happens after a finite execution of the program. Thus, for each violation there is a finite word, also referred to as a *bad prefix*, that witnesses the violation. A *monitor* is a finite state automaton that recognizes bad prefixes.

Monitors have been studied extensively and there are several implementations that use them for model-checking. In [KV01a] Kupferman and Vardi show a methodology for checking safety properties in the case when φ is given as a formula. In [SRA03] Sen, Roşu and Agha show a $c2^{O(2^m)}$ algorithm for constructing the optimal monitor for a safety formula φ, where m is the length of the formula. In contrast to these two approaches, this thesis focuses on the case when the property φ is given as an NFA and we would like to construct the minimal deterministic finite automaton (DFA) that corresponds to φ (recall that this is the *canonical* DFA). The results of the work presented here can be immediately applied to model-checking tools like FoCs [ABG+00]. FoCs converts the property that we want to verify into an NFA, from which the minimal DFA is derived, which in turn is translated into a VHDL[1] process.

The canonization problem is interesting not only from a model-checking point of view, but also from a purely automata-theoretical perspective [HU79, Wat93]. The two leading algorithms for its solutions are by Hopcroft [Hop71], which has the best asymptotic worst-case complexity for the minimization step, and by Brzozowski [Brz62], which is fundamentally different than most other canonization algorithms.

[1]Very high speed integrated circuit Hardware Description Language

While the complexities of Brzozowski's and Hopcroft's algorithms are known [Wat93], there has not been a systematic empirical comparison between them. (A superficial evaluation in [GG97] claims superiority of Hopcroft's algorithm.) In this work I present a direct comparison of the two algorithms and show that none dominates the other for all types of automata.

Chapter 2

The Random Model

2.1 Motivation

The asymptotic complexity of an algorithm does not tell the whole story about its performance. On one hand the asymptotic complexity hides information about the constant factors, and on the other it typically gives the worst-case complexity which may be different from the expected, "in practice" performance. One of the most stunning examples of the discrepancy between theoretical and average case complexity is probably SAT, a decision problem about satisfiability of propositional formulas. SAT is an NP-Complete problem that we can solve in EXPTIME, but by choosing good heuristics one can achieve exponentially better performance. It is a long established tradition to compare the performance of different SAT-solvers on a set of benchmarks.

However, currently there are no existing benchmarks for automata-theoretic algorithms. One possible source could be automata that are used in the industry to perform model-checking. Unfortunately, in the industry the program automaton A_P and the property automaton A_φ are both carefully guarded secrets. Furthermore, one wants to be sure that the automata are general enough, otherwise the performance of one algorithm may be contingent on properties of a particular type of benchmarks. Finally, industrial benchmarks typically have a fixed size, and thus it is very hard to obtain automata with increasing size but similar structural properties, e.g. the same density of accepting states.

For these reasons here I propose a probabilistic model for testing automata-theoretic algorithms. Using such a model gives us several advantages:

- *Generable*: we can generate as many automata as needed.
- *Scalable*: we can generate automata with increasing number of states, while maintaining their structural properties.
- *Generic*: randomly generated automata are not biased toward a particular algorithm.

One question for any probabilistic model is "How realistic is it?". It is impossible to answer it neither positively nor negatively without access to proprietary information. However, in this chapter I will show that the random model allows for a range of "interesting" behaviors for fixed-size and scalability benchmarking. Based on this, I argue that the random model gives us a way of generating different types of automata. In the subsequent chapters I use this model to analyze the performance of the canonization and universality checking algorithm presented in this work.

2.2 Definition

Let $A = (\Sigma, S, S^0, \rho, F)$ be a finite non-deterministic automaton (NFA), where Σ is a finite nonempty alphabet, S is a finite nonempty set of states, $S^0 \subseteq S$ is a non-empty set of initial states, $F \subseteq S$ is the set of accepting states, and $\rho \subseteq S \times \Sigma \times S$ is a transition relation. Recall that A has a canonical, minimal deterministic finite automaton (DFA) that accepts the same language [HU79]. The *canonization* problem is to generate this DFA. A is said to be *universal* if it accepts Σ^*. The *universality problem* is to check if A is universal.

This work proposes a model for building random automata with certain structural properties. In this model the set of initial states S^0 is the singleton $\{s_0\}$ and the alphabet Σ is the set $\{0, 1\}$. For each letter $\sigma \in \Sigma$, a random directed graph $D_\sigma = (S, E_\sigma)$ is generated. The set of vertices of the graph corresponds to the set of states of the automaton, and the set of edges corresponds to the transitions on letter σ in the automaton. More formally, $\rho(s, \sigma, s')$ if and only if $E_\sigma(s, s')$.

Hereafter, the ratio $r_\sigma = \frac{|E|}{|S|}$ is referred to as the *transition density for* σ (intuitively, r_σ represents the expected outdegree of each node for σ). In the model presented here the transition density of D_0 and D_1 is the same, and is referred to as the transition density of A. The idea of using a linear density of some structural parameter to induce different behaviors has been quite popular lately, most notably in the context of random 3-SAT [SML96].

The model for D_σ presented here is closely related to Karp's model of random directed graphs [Kar90]: for each positive integer n and each p with $0 < p < 1$, the sample space consists of all labeled directed graphs $D_{n,p}$ with n vertices and edge probability p. Karp shows that when n is large and np is equal to a constant greater than 1, it is very likely that the graph contains one large strongly connected component and several very small components. When $np < 1$, the expected size of the set of reachable nodes is very small.

It is known that random graphs defined as in [Kar90] in terms of their edge probability or defined as here in terms of the number of edges display essentially the same behavior [Bol01]. Thus, Karp's $np = 1$ corresponds to density 1 in the model presented here. While Karp's considers reachability, which would correspond to non-emptiness [HU79], this work considers canonization and universality. Karp's phase transition at density 1 seems to have no effect on either canonization or universality. The density of the directed graphs underlying the automata here is $2r$, but no interesting phenomenon can be seen at $r = 0.5$.

In the random model the number of final states m is also a linear function of the total number of states, and it is given by a *final state density* $f = \frac{m}{|S|}$. The final states themselves are selected randomly, except for the initial state, which is always chosen to be an accepting state[1]. This additional restriction avoids the cases when an automaton is trivially non-universal because the initial state is not accepting. (One may also consider a model with a fixed number of accepting states rather than with a linear density; we found that such a model behaves similarly to the one we consider

[1] I thank Ken McMillan, the author of `Cadence SMV`, for this suggestion.

here).

2.3 Random Models with Locality

One of the contributions of this work is a comparison of a symbolic and an explicit approach to universality checking. Symbolic algorithms are usually very well suited for discovering and exploiting the underlying structure of a model, but in the random model presented in the previous section there is no structure. This raises the question whether the random model gives unfair advantage to the explicit algorithms. I address this issue by considering two structured random models, described below. Both models restrict the transition relation by localizing the communication between nodes.

2.3.1 Random Model with Grid Structure

On an actual chip the latches are laid out in a rectangular structure where each latch is connected only to a small number of latches in its neighborhood. The *random model with grid structure*, or *grid model* for short, is an abstraction of this configuration. Intuitively, we imagine that the nodes of the automaton are arranged in a square grid and there can only be transitions between nearby nodes. The distance metric is defined using the L_1 distance (also known as the *Manhattan distance*) between the nodes.

More formally, the size of the model is restricted to a perfect square, that is,

$|S| = l^2$ for some natural number l. For each node s_i (where $0 \leq i \leq (l^2 - 1)$ we define two coordinate labels $s_{i_x} = \lfloor i/l \rfloor$ and $s_{i_y} = i \pmod{l}$. The neighborhood is defined using a distance parameter d: a node s_i can make a transitions to s_j only if $|s_{i_x} - s_{j_x}| + |s_{i_y} - s_{j_y}| \leq d|S|$. The accepting states are chosen, as before, at random.

2.3.2 Random Model with Linear Structure

The *random model with linear structure*, or *linear model* for short, assumes that the latches are arranged on a line. As in the grid model, communication is allowed only between nearby nodes. The automaton corresponding to this model is built by restricting the transition relation as follows: a node s_i can make a transition to node s_j, where $0 \leq i, j \leq |S| - 1$, only if $|i - j| \leq d|S|$, where d is the distance parameter. Once again, the accepting states are chosen at random.

2.4 Evaluation of the Random Model

2.4.1 Probability of universality

Figure 2.1 presents the probability of universality as a function of r and f. For each data point 200 (unstructured) random automata with size $|S| = 100$ were generated and checked for universality. The behavior here is quite intuitive. As transition and acceptance densities increase, the automaton has more accepting runs and is therefore more likely to be universal. Note that even if all states are accepting ($f = 1$), the automaton is still not guaranteed to be universal. This follows from the fact that the

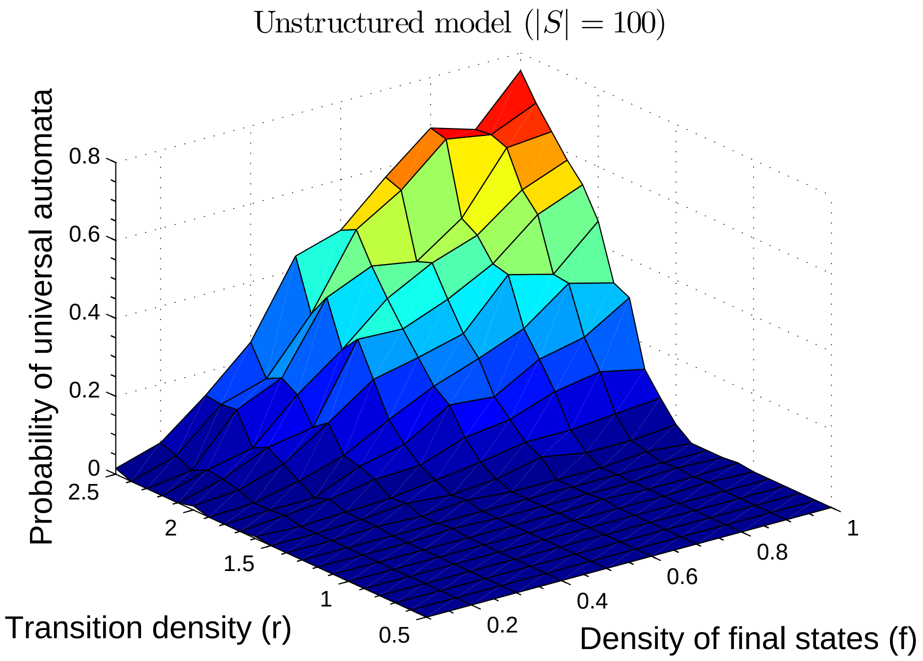


Figure 2.1: Probability of universal automata as a function of transition density r and final state density f, for fixed size ($|S| = 100$) unstructured automata.

transition relation is not necessarily total and the missing transitions are replaced by an implicit transition to a rejecting sink state.

Adding locality does not change the behavior of the model. However, due to the additional restrictions on the transition relation, universal automata appear with lower probability. The effect is more pronounced for automata with matrix structure, shown on Figure 2.2. Figure 2.3 shows the probability distribution for automata with linear structure.

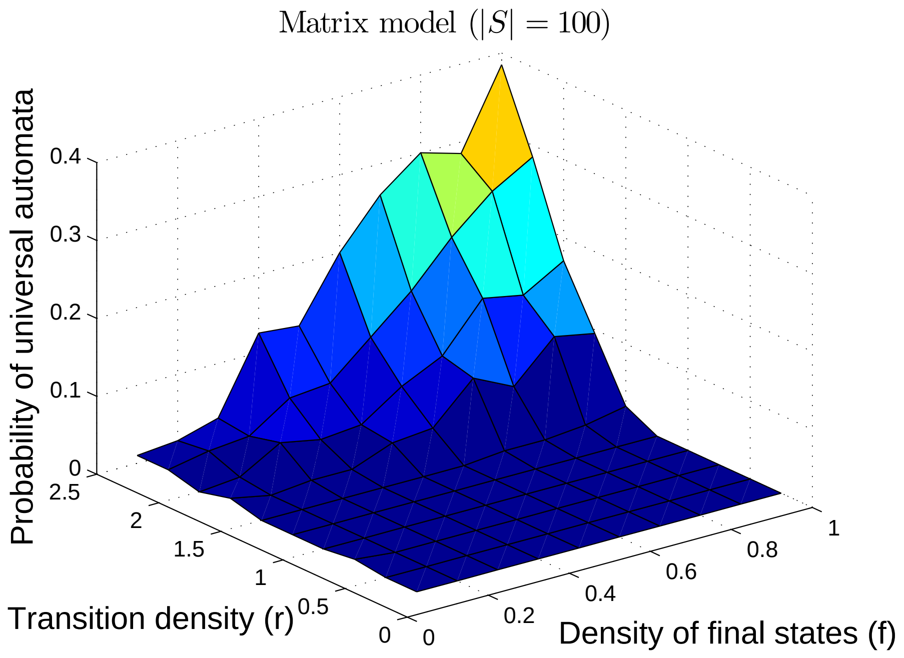


Figure 2.2: Probability of universal automata as a function of transition density r and final state density f, for fixed size ($|S| = 100$) automata with *matrix* structure.

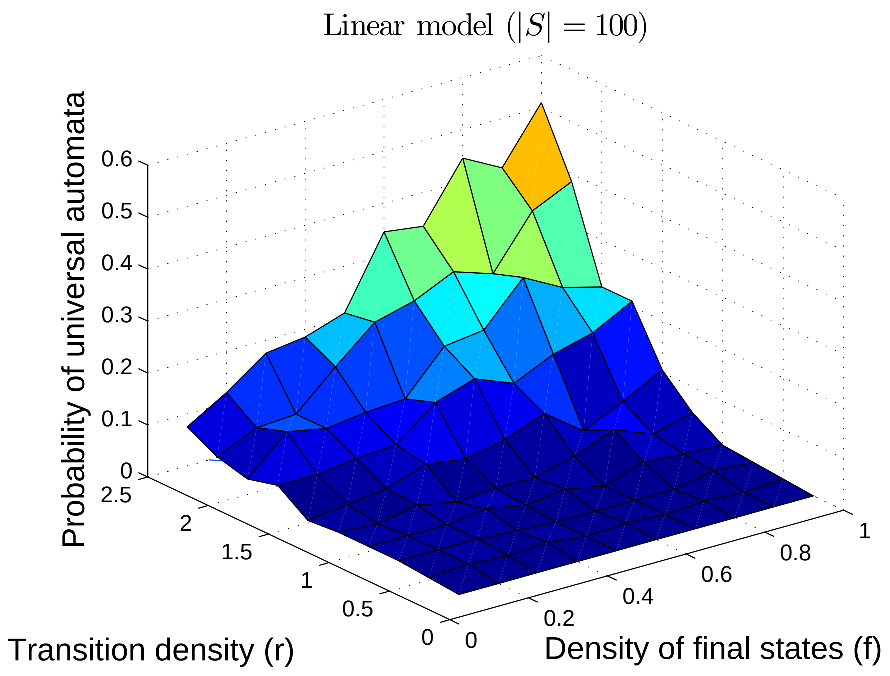


Figure 2.3: Probability of universal automata as a function of transition density r and final state density f, for fixed size ($|S| = 100$) automata with *linear* structure.

2.4.2 Size of the canonical automaton

A completely different pattern emerges when we look at the size of canonical minimized DFA A^C corresponding to the input NFA A (Figure 2.4). For each data point on the graph, 200 random automata were determinized and minimized and then the median of the size of the minimized DFA was taken. I refer to the latter as the *canonical size*. (The motivation for reporting the median rather than the mean is that the median is less affected by outlying points). While the effect of the acceptance density on the canonical size is not too dramatic, transition density does have a dramatic effect on the canonical size. The latter rises and then falls with tradition density, peaking at $r = 1.25$. We see that the canonical size has a coarse phase transition at that density.

2.4.3 Scaling of the size of the canonical automaton

Finally, I show how the canonical size *scales* with respect to the size of the input NFA A. Since the values of f do not have a large effect on the canonical size, for this experiment the density of accepting states was fixed at $f = 0.5$. Figure 2.6 shows that canonical size scales differently at different transition densities. The scaling curves exhibit a range of behaviors. For $r \leq 1.25$ they grow super-polynomially but sub-exponentially (in fact, a function of type $ab^{\sqrt{|S|}}$ provides a very good approximation), for $r = 1.5$ the growth is polynomial, and for higher transition densities they remain almost constant. Interestingly, though in the worst case the canonical size may scale

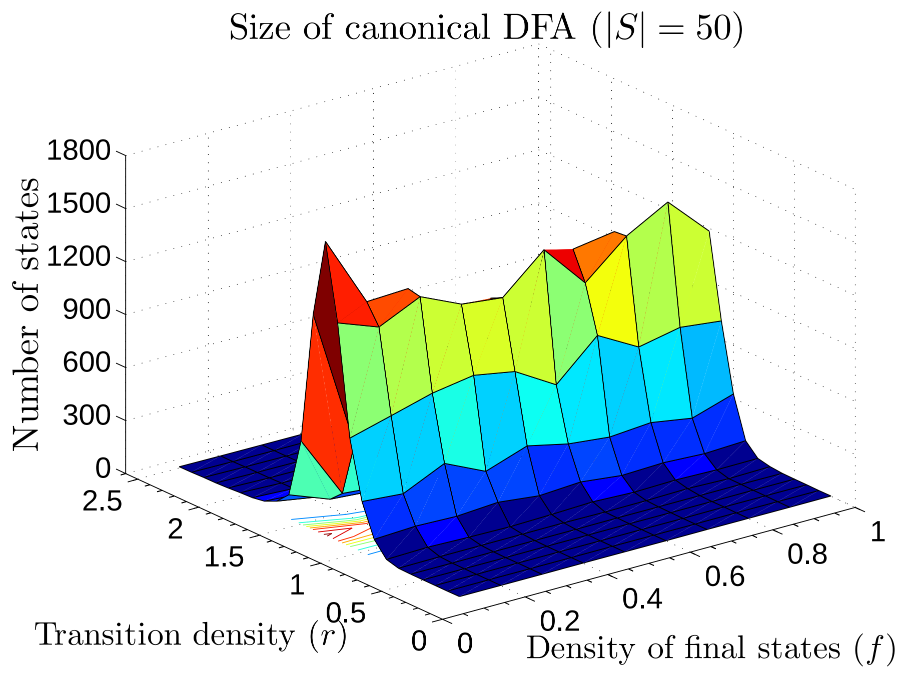


Figure 2.4: Median number of states in the canonical DFA, as a function of transition density r and final state density f, for fixed size ($|S| = 50$) automata.

exponentially [MF71], such exponential scaling is not observed in this probabilistic model.

2.5 Conclusion

The model proposed in this work allows for a range of behaviors as we vary r, f, and the size of the input NFA. It allows one to investigate sets of NFAs which range from almost surely non-universal to almost surely universal. The size of the corresponding canonical DFAs exhibits a phase transition, thereby suggesting that we look for interesting behavior around the transition density of the peak. The size

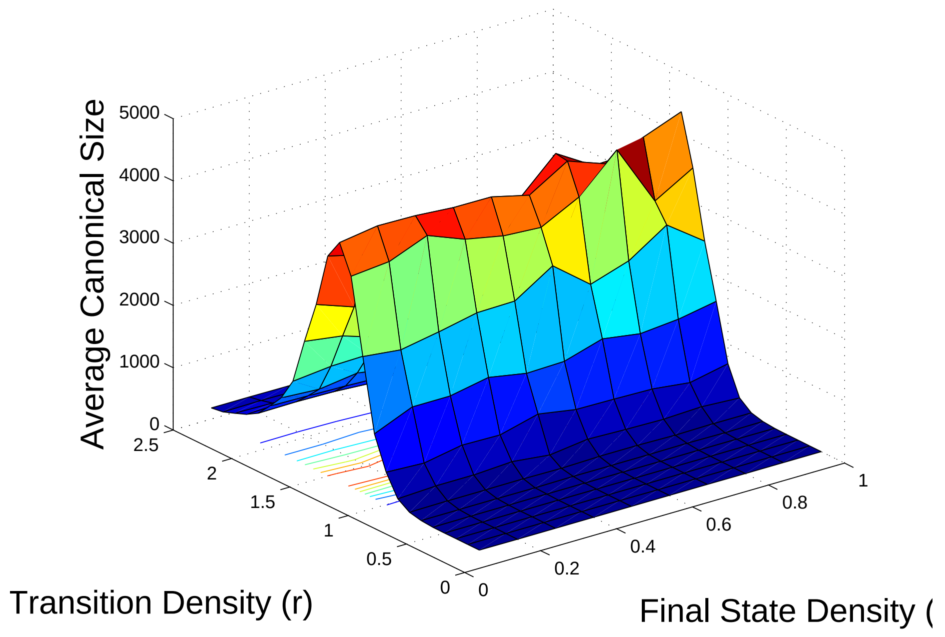


Figure 2.5: Average number of states in the canonical DFA, as a function of transition density r and final state density f, for fixed size ($|S| = 50$) automata.

of the canonical DFAs shows different scaling behavior depending on the transition density, ranging from linear to super-polynomial but sub-exponential. Finally, adding structure to the automata changes the probability of universality, but not the general behavior of the model.

Based on this observation I argue that the random model is a good model to study the performance of automata-theoretic algorithms in practice. The following two chapters present an analysis of the canonization and minimization algorithms using this model.

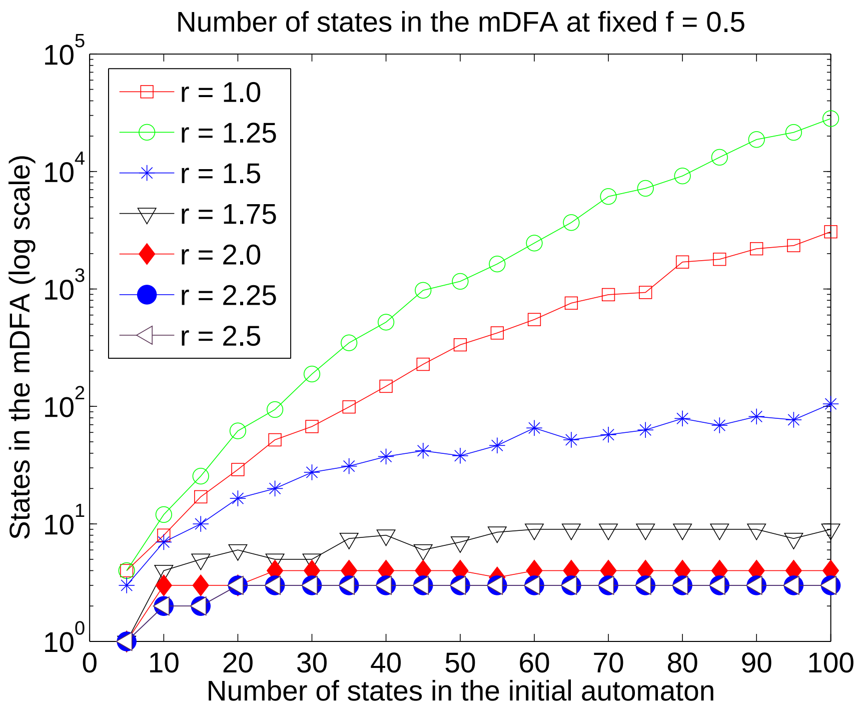


Figure 2.6: Scaling of canonical size at different transition densities (log scale)

Chapter 3

Universality

3.1 Explicit Approach

The straightforward way to check for universality of an NFA $A = (\Sigma, S, S^0, \rho, F)$ is to determinize it, using the subset construction, and then to verify that every reachable state is accepting and that the transition relation is total. Recall that the subset construction returns a deterministic automaton $A^d = (\Sigma, 2^S, \{s_0\}, \rho^d, F^d)$ where

- $\rho^d(T_1, a, T_2) \iff T_2 = \{t_2 \in S : \rho(t_1, a, t_2) \text{ for some } t_1 \in T_1\}$, and
- $F^d = \{T \in 2^S : T \cap F \neq \emptyset\}$.

Notice that in this approach the sets of states (T_1 and T_2 in the above definition) have to be maintained explicitly, hence it is referred to as the *explicit approach*. In this work I use two distinct methods for solving the universality problem explicitly. One of them is using a Java tool called `Automaton.brics.dk` [Mø04], and the other–using the model-checker `SPIN` [Hol97, Hol04].

Although both `Automaton` and `SPIN` can check for universality explicitly, there are key differences between them. `Automaton` is optimized for manipulating automata and this gives it an advantage over `SPIN` which is a general purpose model-checker. More importantly, the symbolic approach is implemented via `SMV`, another model-checker, and we would like to compare the performance of tools with similar expressive power.

3.1.1 `Automaton.brics.dk`–based method

`Automaton.brics.dk` (hereafter referred to as `Automaton`) is a Java tool for manipulating automata. It implements the subset construction by creating the subsets (states of A^d) on-the-fly, as they are reached. Notice that for the universality problem one does not need to generate the whole automaton A^d a priory and then check for universality.

As an optimization, each subset can be checked for universality at the time it is first reached. If the subset is not accepting (that is, it does not contain a state from F), then A^d is non-universal and the algorithm terminates early. Intuitively, this corresponds to a on-the-fly search for a non-accepting state in A^d. In particular, since `Automaton` maintains the reached states in a queue, this corresponds to a breadth-first search.

3.1.2 `SPIN`–based method

`SPIN` is a model-checker implemented in `C`. `SPIN` allows the specification of concurrent systems using a high-level language called `Promela`. The state of the system is maintained explicitly using a underlying automaton, and similar to `Automaton`, `SPIN` works on-the-fly, without constructing the state-space a priori.

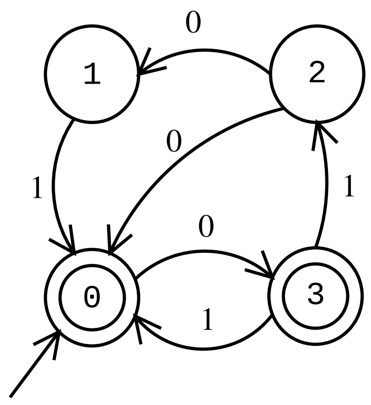
The key to encoding the automaton in a model-checker is the observation that A^d could be viewed as a *sequential circuit* (SC). An SC [HS96] is a tuple (I, L, δ, α) where I is a set of input signals, L is a set of registers, $\delta : 2^L \times 2^I \rightarrow 2^L$ is the next-state function, describing the next assignment of the of the registers given their current assignment and an input assignment, and $\alpha \in 2^L$ is an initial assignment to the registers. (Usually we also have output signals and an output function, but this is not needed here.) The alphabet $\Sigma = \{0, 1\}$ corresponds here to a single input signal. The state set S can be viewed as the register set L; a set in 2^S can be viewed as a Boolean assignment to the state in S, using the duality between sets and their characteristic functions. The intuition is that every state in S can be viewed as a register that is either "active" or "inactive". The initial state s_0 correspond to an initial assignment, assigning 1 to s_0 and 0 to all other registers, as only s_0 is active, initially. Finally, the transition relation ρ_d, which is really a function, correspond to the next-state function, where $\delta(P, \sigma) = Q$ when $\rho_d(P, a, Q)$ holds (note that we view here subsets of S as Boolean assignments to S). Universality of A now correspond to an invariance property of A_d saying that at each step at least one of the registers corresponding to a final state is high.

Next I present the encoding of the universality problem into `Promela`. The model consists of two concurrent processes. One process (the "driver") non-deterministically

```
active proctype driver() {
  do
  :: letter = true;
  :: letter = false;
  od;
}
```

Figure 3.1: `Promela` representation of the process that "drives" the automaton by selecting the transition letter.

selects the next transition letter (Figure 3.1). The other process (the "model") defines the transitions of the automaton. The model maintains two boolean vectors of size $|S|$, one corresponding to the superstate (set of states of the automaton) that it is currently visiting (`state[]`) and the other–to the next reachable superstate (`nextstate[]`). At first only the initial state of the automaton is active, therefore the superstate vector `state[]` is equal to $10^{|S|-1}$. For each state $s \in S$ and for each letter $\sigma \in \Sigma$ the model checks if s is active in the current subset and if σ is the transition letter selected by the driver. If this is the case and if A has an outgoing transition from s on σ to a set of states $\{s_1, s_2, \ldots, s_n\}$, the `Promela` model activates the states corresponding to $\{s_1, s_2, \ldots, s_n\}$ in `nextstate[]`. Finally, the universality condition is encoded as an assertion that at least one of the accepting states of A is active in every reachable subset. Figure 3.2 illustrates the main parts of the encoding for a particular automaton. The full encoding is presented in Appendix A.



```
active proctype model() {
  bool state[4] = false;
  bool nextstate[4] = false;
 (...)
  if
    :: (state[0] && letter) -> skip;
    :: (state[0] && ! letter) -> nextstate[3] = true;
    :: else -> skip;
  fi;
 (...)
  if
    :: (state[2] && letter) -> skip;
    :: (state[2] && ! letter) -> nextstate[0] = true;
                                 nextstate[1] = true;
    :: else -> skip;
  fi;
 (...)
  assert ( nextstate[0] || nextstate[3] );
}
```

Figure 3.2: Partial `Promela` encoding of the transitions of the automaton.

3.2 Symbolic Approach

As an alternative to the explicit approach this work also considers a symbolic approach to solving the universality problem. At the heart of this approach lie binary decision diagrams (BDDs). First I give a brief introduction to BDDs and then describe the encoding of the problem into a model for two BDD-based model-checkers.

3.2.1 Binary Decision Diagrams

Binary decision diagrams, introduced by Bryant in [Bry86, Bry92], provide an efficient way of representing and manipulating Boolean functions. A BDD (see Figure 3.3) is a rooted, directed acyclic graph with one or two terminal nodes labeled **0** or **1**, and a set of variable nodes of out-degree two. An ordered binary decision diagram (OBDD) is a BDD in which the variables respect a given linear order on all

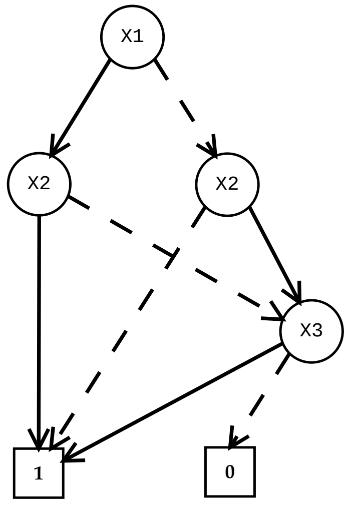


Figure 3.3: OBDD encoding of the function $((x_1 \iff x_2) \vee x_3)$

paths from the root to a leaf. The order in which the variables appear can affect the size of the OBDD, and in some cases can make the difference between a linear and exponential representation. Choosing an optimal variable order is an NP-complete problem, and one has to resort to various heuristics. For a given order of variables, OBBDs provide a canonical representation.

Each path from root to leaf represents an assignment to each of the variables on the path; traditionally, solid edges represent an assignment of *true*, and the dashed– and assignment of *false* (Figure 3.3). Since there can be exponentially more paths than vertices and edges, OBDDs are often substantially more compact explicit representations, and have been used successfully in the verification of complex circuits [BCM+92].

3.2.2 `SMV` Encoding of the Universality Problem

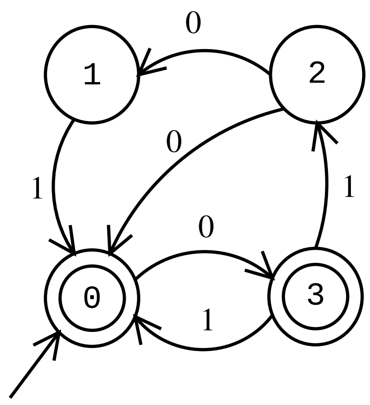
In order to analyze the performance of the symbolic approach I used two symbolic model checkers, both referred to as `SMV`: `Cadence SMV` [McM93] and `NuSMV` [CCG$^+$02]. The encoding of the problem is based again on the observation made in Section 3.1.2 that an automaton can be viewed as a sequential circuit (SC).

To encode the SC for A^d in `SMV`, we use the states in S as state variables, corresponding to the registers. The SC is defined via the `init` and `next` statements: `init(s)=1` iff $s = s_0$, and `next(s)=1` iff $\bigvee_{\rho(t,\sigma,s)} t$, when the input is σ (see Figure 3.4 for an example). A single Boolean vector encodes the registers, and a variable `input` encodes the input symbol. Note that in contrast to the `Promela` encoding presented in Section 3.1.2, here we don't need a driver for the input symbol; leaving `input` unconstrained forces `SMV` to assign values to the variable non-deterministically. Universality of A correspond to a safety property of A^d which states that at all times one or more of the registers corresponding to the final states are set to 1. This property is expressed in CTL as $AG(\bigvee_{s \in F} s)$.

3.2.3 Optimizations

In order to improve the running time of `NuSMV` and `Cadence SMV`, several optimization techniques were considered.

Sloppy vs. fussy encoding This optimization is based on the observation that to check universality we need not determinize A. Instead of encoding A^d, we can encode



```
MODULE main
VAR
  state: array 0..3 of boolean; input: boolean;

ASSIGN
  init(state[0]) := 1; init(state[1]) := 0;
  init(state[2]) := 0; init(state[3]) := 0;

next(state[0]) := ((state[1] & input) |
      (state[2] & ! input) | (state[3] & input));
next(state[1]) := (state[2] & ! input);
next(state[2]) := (state[3] & input);
next(state[3]) := (state[0] & ! input);

SPEC
  AG ( state[0] | state[3] );
```

Figure 3.4: A simple automaton and its encoding in SMV

the non-deterministic automaton $A^n = (\Sigma, 2^S, \{s_0\}, \rho^n, F^d)$, where

- $F^d = \{T \in 2^S : T \cap F \neq \emptyset\}$, (as before), and
- $\rho^n(T_1, a, T_2) \iff T_2 \supseteq \{t_2 \in S : \rho(t1, a, t_2)$ for some $t_1 \in T_1\}$.

I will call this a *sloppy encoding* to distinguish it from the A^d-based encoding, which will be referred to as *fussy encoding.* The sloppy encoding satisfies the `SMV` specification exactly when the fussy encoding satisfies it. Intuitively, A_n allows more superstates to be active in the subset construction. If a non-accepting superstate (subset of S) is reachable in A^d, then the same superstate will be reachable in A^n. Likewise, if all reachable superstates in A^d are accepting (i.e. contain an element of F), then all reachable superstates in A^n will contain an element of F (recall that the superstates in A^n are supersets of the superstates in A^d) and will also be accepting. This argument is formalized below.

Theorem. *Let A^n and A^d be defined as above. Then a non-accepting superstate is* *reachable in A^n iff a non-accepting superstate is reachable in A^d.*

Proof. Suppose that a non-accepting superstate $q_m \in 2^S$ is reachable in A^d. Then there exists a word $a_0 a_1 \dots a_{m-1}$ and a set of superstates $q_0, q_1, \dots, q_{m-1} \in 2^S$ such that $q_0 = \{s_0\}$ and $\rho^d(q_i, a_i, q_{i+1})$ for $0 \leq i \leq m-1$. By the definition of A^n, the initial superstate is q_0 and for each q_i there is a transition from q_i to q_{i+1} on a_i. Thus, $q_0, q_1, \dots, q_{m-1}, q_m$ are reachable superstates in A^n, and in particular, q_m is reachable. Moreover, q_m is not accepting in A^d so it cannot be accepting in A^n, therefore a non-accepting state is reachable in A^n.

To prove the theorem in the other direction, suppose that a non-accepting superstate is reachable in A^n, and we have to show that a non-accepting superstate is reachable in A^d. Using the contrapositive, we have to show that if all reachable states in A^d are accepting, then all reachable states in A^n are accepting.

Suppose that we run both A^d and A^n on the same word $w \in \Sigma^*$. Because A^d is deterministic, it will reach a uniquely determined superstate $q = \{s_{i_1}, s_{i_2}, \dots, s_{i_k}\}$. A^n, on the other hand, is non-deterministic, so it may potentially reach more than one superstate. We will show that no matter what state $q' = \{s_{j_1}, s_{j_2}, \dots, s_{j_p}\}$ it reaches, it is always the case that $\{s_{i_1}, s_{i_2}, \dots, s_{i_k}\} \subseteq \{s_{j_1}, s_{j_2}, \dots, s_{j_p}\}$. Then the theorem follows trivially: since every reachable superstate in A^d is accepting, at least one of $s_{i_1}, s_{i_2}, \dots, s_{i_k}$ must be an element of F. Since $q \subseteq q'$, it follows that q' also contains an element of F, and is therefore also an accepting superstate. $\square$

Lemma. *Let A^d and A^n be defined as above, and let $w = a_0 a_1 \dots a_m$ be a word.*

Suppose that after consuming w A^d ends up in state $q = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ and A^n in state $q' = \{s_{j_1}, s_{j_2}, \ldots, s_{j_p}\}$. Then $q \subseteq q'$.

Proof. The proof is by induction on the length of w.

Base case: If $|w| = 0$ then both A^d and A^n will be in their initial superstates, which are the same set $\{s_0\}$, so the lemma trivially holds.

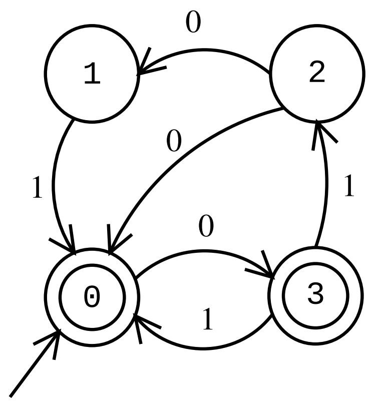
Inductive case: Suppose that the lemma holds for $0 \leq |w| \leq m$. Let $q_0, q_1, \ldots, q_{m+1}$ and $q'_0, q'_1, \ldots, q'_{m+1}$ be subsets of 2^S such that $q_0 = q'_0 = \{s_0\}$, $\rho^d(q_i, a_i, q_{i+1})$ and $\rho^n(q'_i, a_i, q'_{i+1})$ for $0 \leq i \leq m$. By inductive hypothesis $q_m \subseteq q'_m$.

By definition,

$$\begin{array}{rcl} q'_{m+1} & \supseteq & \{t'_2 \in S : \rho(t'_1, a, t'_2) \text{ for some } t'_1 \in q'_m\} \\ & \supseteq & \{t_2 \in S : \rho(t_1, a, t_2) \text{ for some } t_1 \in q_m\} \\ & = & q_{m+1} \end{array}$$

This completes the proof. □

Unlike A^d, we cannot view A^n as a sequential circuit, since it is not deterministic. `SMV`, however, can also model non-deterministic systems. Rather than requiring that `next(s)=1` **iff** $\bigvee_{\rho(t,\sigma,s)} t$, when the input is σ, we requite that `next(s)=1` **if** $\bigvee_{\rho(t,\sigma,s)} t$, when the input is σ (the "iff" is replaced by "if"). In an explicit construction the sloppy approach would generate more subsets, but in a symbolic approach the sloppy approach uses "looser" logical constraints (`trans`, rather than `assign`), which might



```
MODULE main
VAR
  state: array 0..3 of boolean; input: boolean;

ASSIGN
  init(state[0]) := 1; init(state[1]) := 0;
  init(state[2]) := 0; init(state[3]) := 0;

TRANS
  (state[0] & (! input)) -> next(state[3]);
  ((state[1] & input) | (state[2] & (! input)) |
               (state[3] & input)) -> next(state[0]);
  (state[2] & (! input)) -> next(state[1]);
  (state[3] & input) -> next(state[2]);

SPEC
  AG ( state[0] | state[3] );
```

Figure 3.5: Sloppy encoding of the same automaton as in Figure 3.4

result in smaller BDDs. See Figure 3.5 for a sloppy encoding of the automaton in Figure 3.4.

Monolithic vs. conjunctive partitioning In [BCL91] Burch, Clarke and Long suggest an optimization of the representation of the transition relation of a sequential circuit. They note that the transition relation is the conjunction of several small relations and the size of the BDD representing the entire transition relation may grow as the product of the sizes of the individual parts. This encoding is called *monolithic*. The method that Burch *et al.* suggest represents the transition relation by a list of the parts, which are implicitly conjuncted. Burch *et al.* call their method *conjunctive partitioning*, which has since then become the default encoding in `NuSMV` and `Cadence SMV`.

Conjunctive partitioning introduces an overhead when calculating the set of states reachable in the next step. The set of transitions has to be considered in some order, and choosing a good order is non-trivial, because each individual transition may depend on many variables. In large systems the overhead is negligible compared to the advantage of using small BDDs [BCL91]. However, in our models the transitions are fairly simple, and it is not immediately clear whether monolithic encoding is a better choice.

Variable ordering When using BDDs, it is crucial to select a good order of the variables (See Figures 3.6 and 3.7) because the order had a huge impact on the size of the BDD. Finding an optimal order is itself a hard problem, so one has to resort to different heuristics. The default order in NuSMV corresponds to the order in which the variables are first declared; in Cadence SMV it is based on some internal heuristic. The orders that were considered included the default order, and the orders given by three heuristics that are studied with respect to tree decompositions: Maximum Cardinality Search (MCS), LEXP and LEXM [KBvH01]. In the experiments MCS proved to be better than LEXP and LEXM, so in this work I will only report the results for MCS and for the default order.

In order to apply MCS we view the automaton as a graph whose nodes are the states, and in which two nodes are connected iff there is a transition between them. MCS [TY84] orders the vertices from 1 to $|S|$ according to the following rule: The first node is chosen arbitrarily. From this point on, a node that is adjacent to a maximal number of already selected vertices is selected next, and so on. Ties can be broken

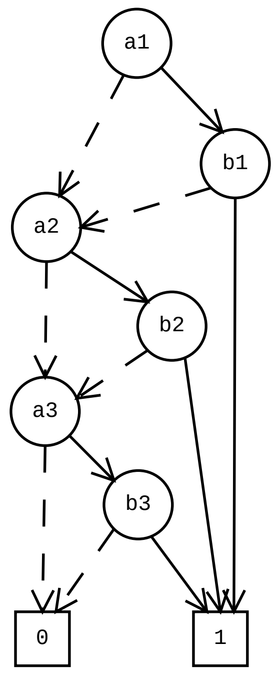


Figure 3.6: Optimal order of the BDD variables for $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

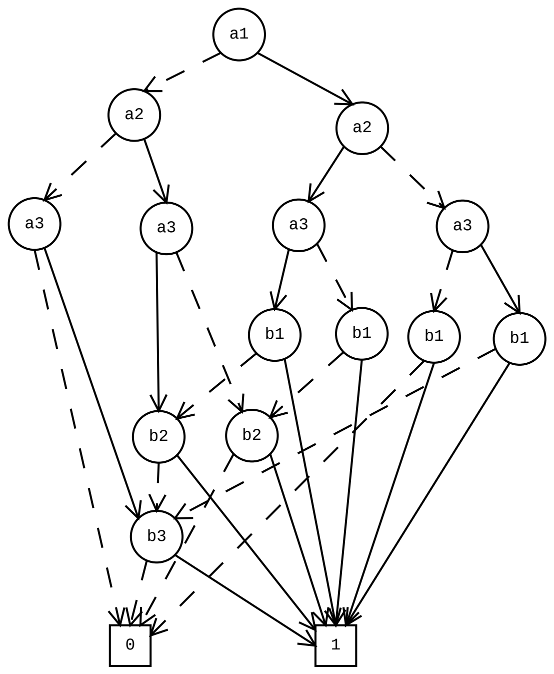


Figure 3.7: Bad order of the BDD variables for $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

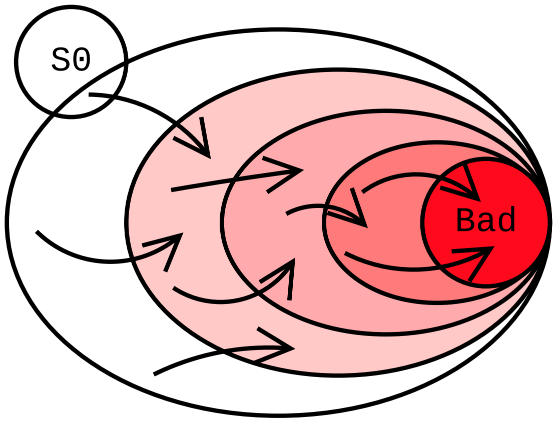
in various ways (eg. minimize the degree to unselected nodes [AV01] or maximize it [BB94], or select one at random), but none leads to a significant speedup. For the experiments presented here, when MCS was used ties were broken by minimizing the degree to the unselected nodes.

Traversal In the `SMV` model the safety condition is of the form $\mathbf{AG}\alpha$: i.e. α is a property that we want to hold in all reachable states. CTL formulas are normally evaluated backward in `NuSMV` [CCGR00], via the greatest fixpoint characterization:

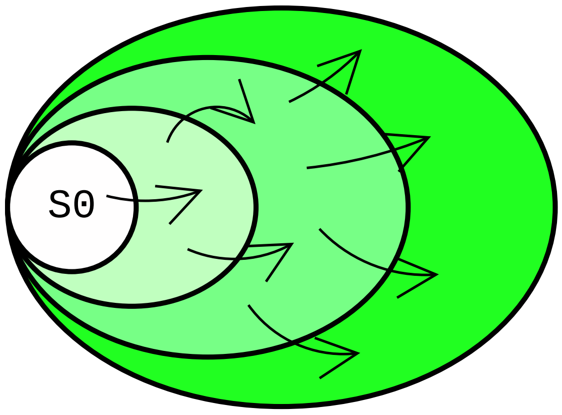
$$\mathbf{AG}\alpha = \mathbf{gfp}_Z[\alpha \wedge \mathbf{AX}Z]$$

This approach ("backward traversal") can be sometimes quite inefficient. As an optimization (only for $\mathbf{AG}\alpha$ formulas), `NuSMV` supports another strategy: calculate the set of reachable states, and verify that they satisfy the property α ("forward traversal") (See Figure 3.8). In `Cadence SMV`, forward traversal is the default mode, but backward traversal is also available. Both model-checkers were run with both traversal modes.

Evaluating The Symbolic Approach Generally, the running times of the various symbolic approaches increase with both transition density and acceptance density. Figures 3.9 and 3.10 present the effect of the first three optimizations to the running times of NuSMV and Cadence SMV for fixed size unstructured automata (in this set of experiments forward traversal was used). No single configuration gives the best performance throughout the range of transition density. Nevertheless, several



(a) Backward traversal



(b) Forward traversal

Figure 3.8: Forward and backward traversal of the model

conclusions can be made about the individual optimizations. Ordering the variables with MCS is always better than using the default ordering. Monolithic encoding is better than conjunctive partitioning for low transition density; the threshold varies depending on the tool and the choices for the other optimizations. Sloppy encoding is better than fussy when used together with monolithic encoding; the opposite is true when using conjunctive partitioning. The only exception to the latter is sloppy monolithic encoding in NuSMV, which gives the worst performance. Overall, for both tools, the best performance is achieved by using *monolithic-MCS-sloppy* up to $r = 1.3$, and *conjunctive-MCS-$\star$* thereafter (the results for sloppy and fussy are too close to call here).

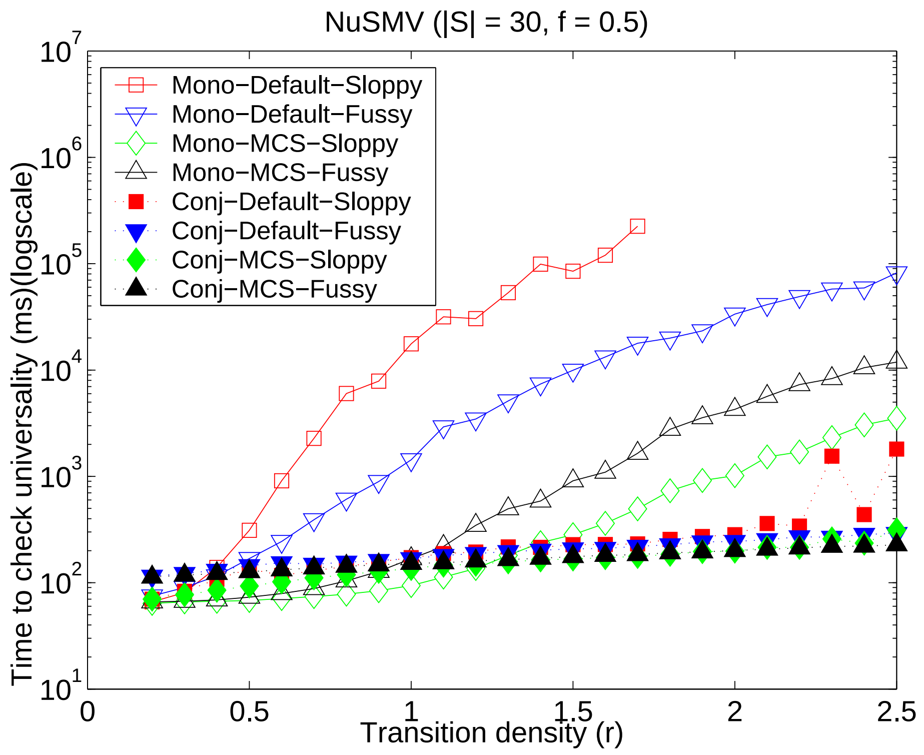


Figure 3.9: NuSMV

In order to fine-tune the two tools I next looked at their scaling performance (Figure 3.11). I first consider automata with $f = 0.9$ and $r = 2.5$ (the choice is explained later). I fix the transition encoding to conjunctive and variable order to MCS, and varied traversal direction and sloppy vs. fussy encoding. For both tools backward traversal is the better choice, not surprisingly, since 90% of the states are accepting and a fixed point is achieved very quickly. When using backward traversal,

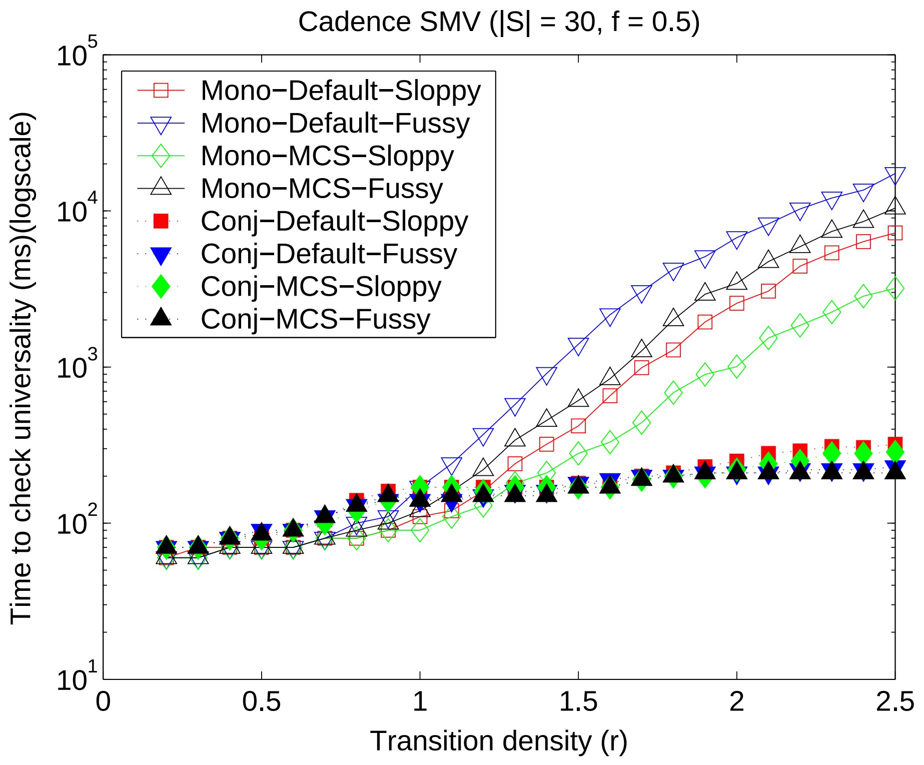


Figure 3.10: Cadence SMV

sloppy encoding gives better performance, and the opposite is true when using forward traversal. Overall, the best scaling is achieved by Cadence SMV with backward traversal and sloppy encoding.

Comparing The Explicit and Symbolic Approaches I first compare the performance of the explicit and the symbolic approaches on a set of random automata with a fixed size. For each data point I took the median of all execution times (200 sample points). All experiments were performed on the Rice Terascale Cluster[1], which is a large Linux cluster of Itanium II processors with 4 GB of memory each.

[1]http://support.rtc.rice.edu/

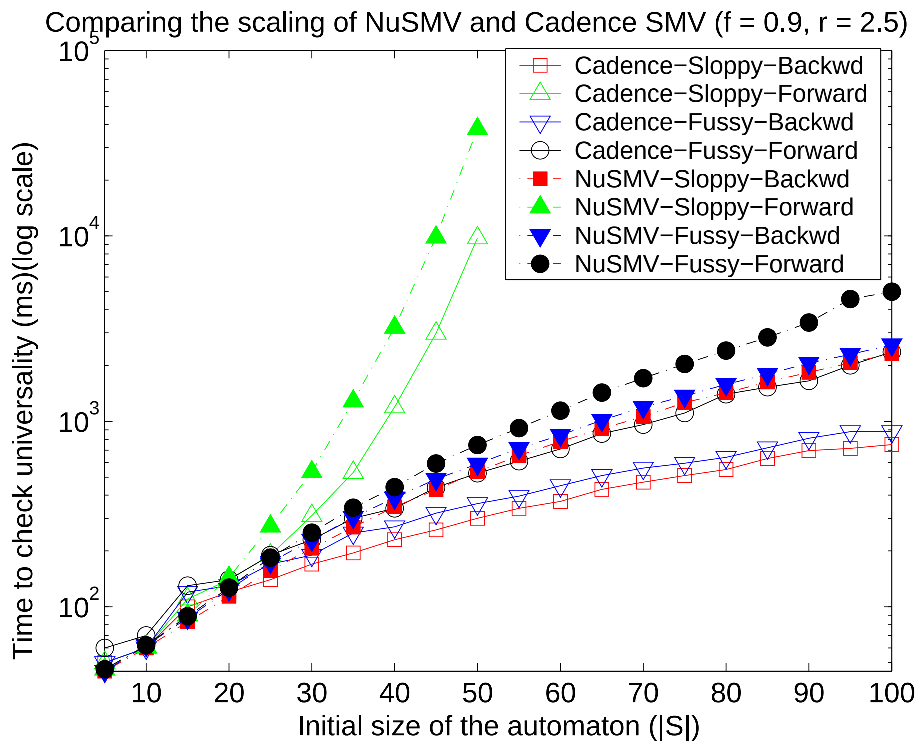


Figure 3.11: Optimizing `NuSMV` and `Cadence SMV` (scaling)

The explicit algorithm here is represented by `Automaton`.

The results indicate that for small automata the explicit algorithm is much faster than the symbolic. In fact, even when using automata with initial size $|S| = 100$, the median of the execution time is 0 almost everywhere on the landscape (see Figure 3.12). In contrast, even for automata with $|S| = 30$ the symbolic algorithm takes non-negligible time (Figures 3.9 and 3.10).

As before, we are also interested which algorithm scales better as the initial size of the automata increases. For this set of experiments, I used unstructured automata with fixed densities of the final states ($f = 0.9$) and the transitions ($r = 2.5$), i.e. on of the furthest edge of the landscape. I chose this point because almost everywhere

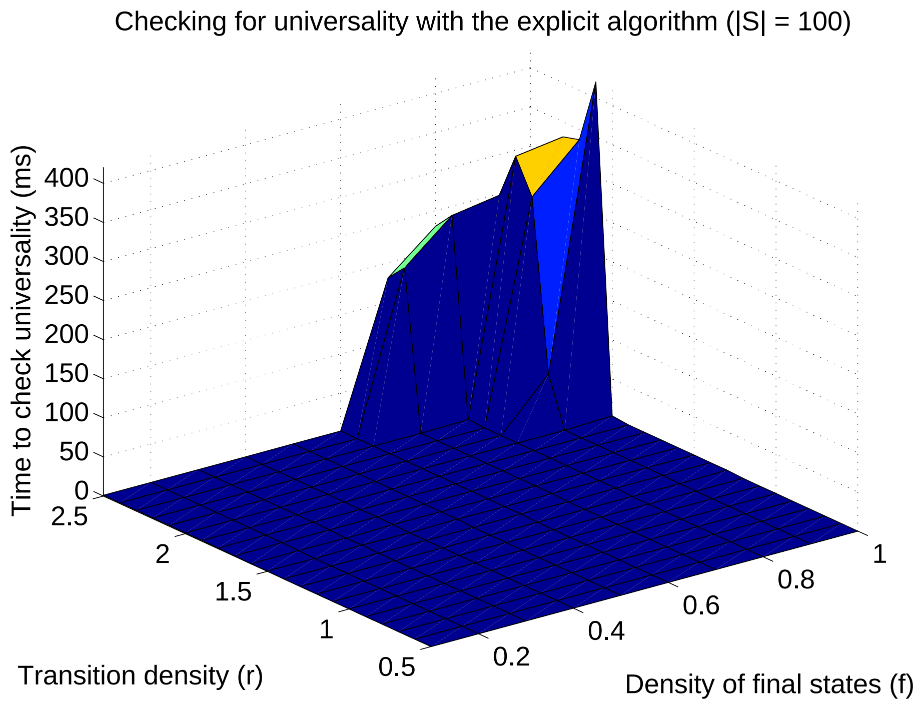


Figure 3.12: Median time to check for universality with the explicit algorithm (using `Automaton`)

else the median execution time of the explicit algorithm implemented in `Automaton` is 0 for small automata. I varied the initial size of the automata between 5 and 600. The results are presented on Figure 3.13. The symbolic algorithm (Cadence SMV) is slower than the explicit (`Automaton`) throughout the whole range. All algorithms scale sub-exponentially; however, the symbolic algorithm scales $2^{O(\sqrt{|S|})}$ worse than the explicit one. I also present data for `NuSMV`, which scales the worst of the three algorithms and is the slowest for $|S| > 20$. Note that at lower transition and/or acceptance density, the advantage of the explicit approach over the symbolic approach is much more pronounced.

One might object that `Automaton` performs better because it is a specialized tool.

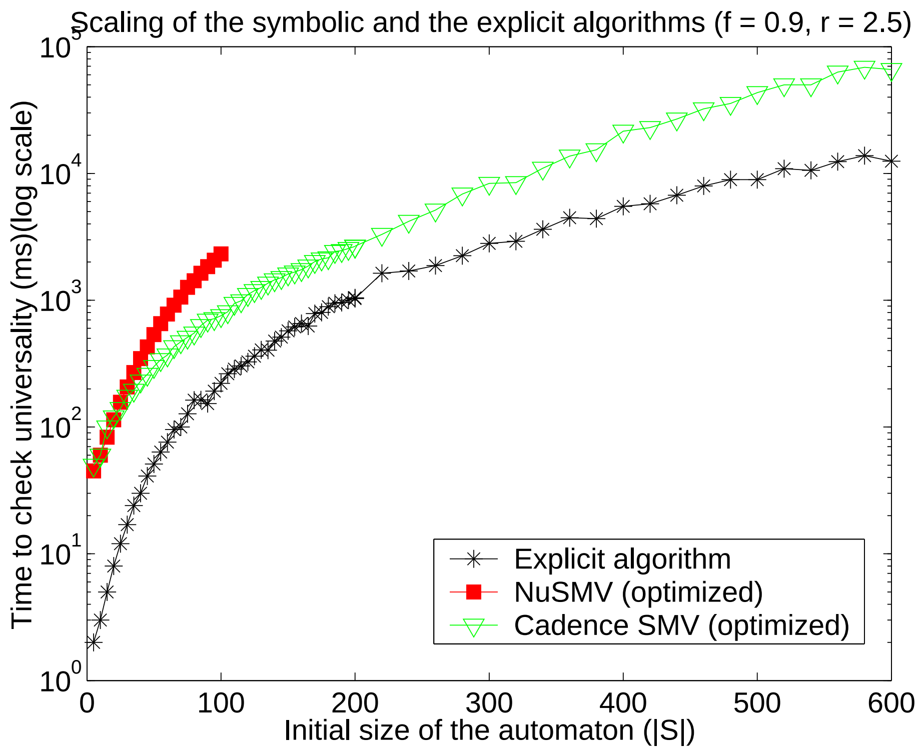


Figure 3.13: Scaling comparison of the symbolic (`SMV`) and explicit (`Automaton`) algorithms: Logarithmic plot

In order to present a full comparison between the symbolic and the explicit algorithms, I also considered the performance of `SPIN`. On Figure 3.14 I present scaling comparison of `Cadence SMV`, `SPIN` and `Automaton`. The direct comparison of the two model-checkers shows that the explicit one (`SPIN`) is much faster than the symbolic one. Plotting the same data in a log-log plot (Figure 3.15 further shows that both explicit algorithms scale better than the symbolic one.)

Comparing The Explicit and Symbolic Approaches on structured automata

One may argue that the symbolic algorithm is performing so poorly because the automata did not have any structure. BDD-based algorithms are usually very successful

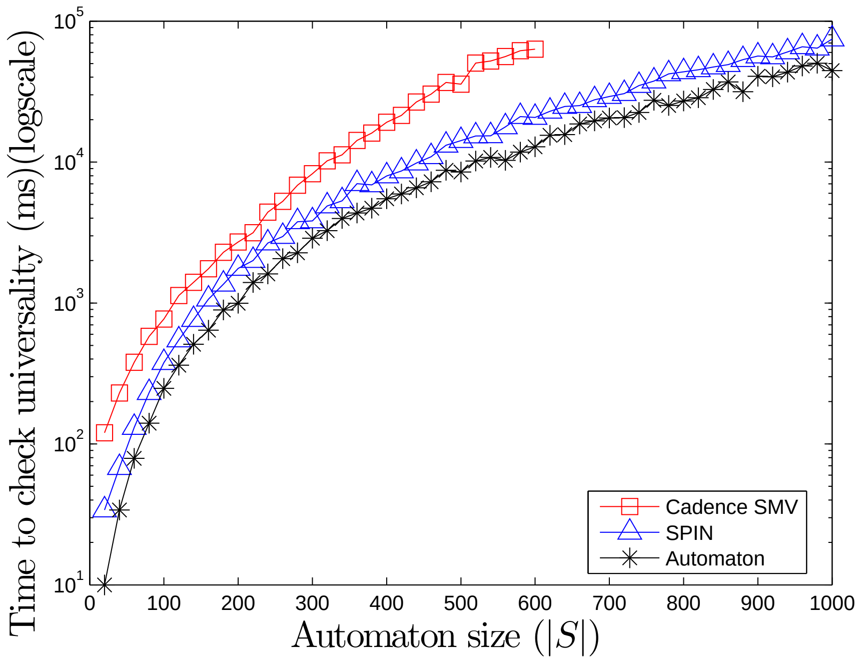


Figure 3.14: Scaling comparison of the symbolic (`Cadence SMV`) and the explicit (`Automaton`, `SPIN`) algorithms: Logarithmic plot

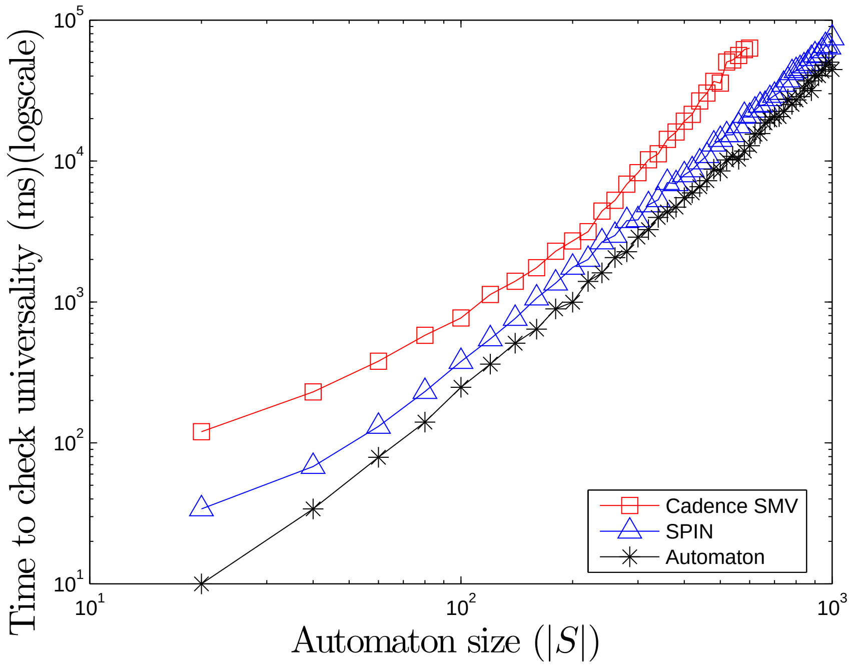


Figure 3.15: Scaling comparison of the symbolic (`Cadence SMV`) and the explicit (`Automaton`, `SPIN`) algorithms: Log-log plot

at discovering and exploiting structures in the model. To gain a fuller understanding of the performance of the symbolic and the explicit algorithm, and to address this issue, I also performed experiments with structured models. I used `Cadence SMV`, `SPIN` and `Automaton` as before, and considered the matrix and linear models of automata. For each of the models I optimized the performance of `Cadence SMV` using the four optimizations presented above. The best execution time for `Cadence SMV` was achieved using conjunctive partitioning, sloppy encoding, MCS order and backward traversal, on both the matrix and the linear models.

Figure 3.16 presents the results for the matrix model, with $r = 2.5$, $f = 0.9$, and distance parameter $d = 0.05$. (The data on this and the subsequent figure is the average rather than the median because there is a bimodal distribution of the execution time.) Each of the three tools showed a speed-up compared to their performance on the unstructured model. This is not surprising given that the structured models are more restrictive: since fewer automata are universal, a counterexample, when it exists, is easier to discover. However, the symbolic algorithm (represented by `Cadence SMV`) is still performing worse than the explicit algorithm (`Automaton`, `SPIN`).

The results for the linear model are presented on Figure 3.17. Unlike the matrix model, the linear model suggests a very nice linear order of the variables. The ordering heuristic, MCS, easily discovers this order and allows for almost optimal size of the BDDs in the symbolic algorithm. Thus, the symbolic algorithm scales better than the

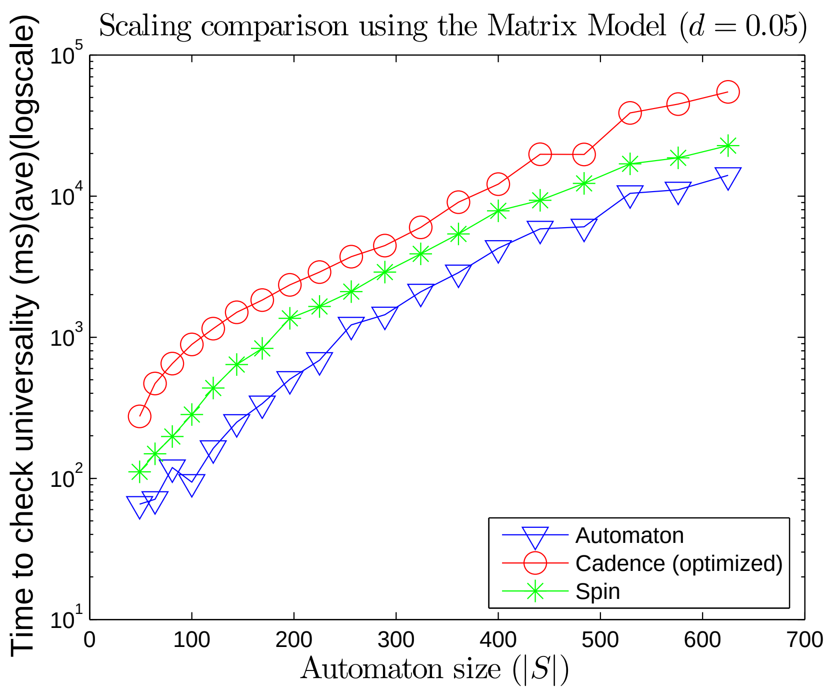


Figure 3.16: Scaling comparison of the symbolic (`Cadence SMV`) and the explicit (`Automaton`,`SPIN`) algorithms: Matrix model

explicit one. Even though this result is encouraging, the linear model is too extreme and less realistic than the matrix model in terms of laying out latches on a chip. Only automata modeling very special circuits will contain a linear structure like the one used here.

3.3 Conclusions

This section presents a comparison between a symbolic and an explicit algorithm for checking universality. The discovery that the explicit approach scales better than the symbolic one is rather surprising. The advantage of using an explicit algorithm is seen even when we compare two model-checkers(`SMV` and `SPIN`). Adding "matrix"

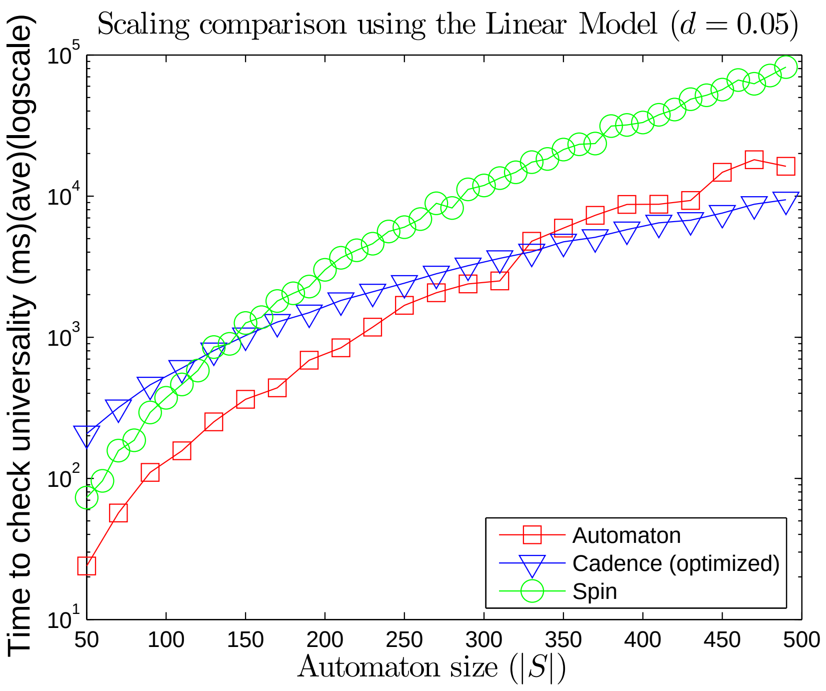


Figure 3.17: Scaling comparison of the symbolic (`Cadence SMV`) and the explicit (`Automaton`,spin) algorithms: Linear model

structure to the model does not change the results, and only after considering the highly-restricted "linear" model can we see the symbolic tool dominating the explicit ones. These results indicate that the reputation of the symbolic algorithms for better handling asynchronous systems may need to be considered more carefully.

Chapter 4

Canonization

This chapter investigates algorithms for deriving the canonical (minimal) DFA corresponding to a given NFA. In addition to its immediate applications in compiler construction and automata theory, the work is motivated by a special type of properties called safety properties, that we might be asked to verify.

4.1 Safety Properties

Intuitively, a safety property asserts that the system always stays within some allowed set of finite behaviors, and every violation occurs after a finite execution [KV99].

Formally, the computations of a non-terminating system can be viewed as an infinite sequence of system states, where each state denotes the set of the atomic propositions holding at that state. We call each of these sequences a *trace*. We use traces to define a safety properties and bad prefixes.

Definition (Safety Property [SRA03]). *A property is a* safety property *if for every infinite trace ρ there exists a finite prefix α such that for all infinite traces ρ', $\alpha \cdot \rho'$ does not satisfy the property. The prefix α is called a* bad prefix.

A *monitor* is a finite automaton (on finite words) that accepts the bad prefixes of a safety property. In the context of model-checking, automata on finite words are usually more helpful because upon a violation of the property they can return a finite trace, while an automaton for a general property may return an infinite trace [KV99].

The *canonization* problem consists of constructing the minimal DFA that accepts the same language as a given (possibly non-deterministic) finite automaton. The next section presents three classical algorithms for producing the canonical automaton.

4.2 Canonization Algorithms

There are two different approaches to canonization. The first approach involves a two-step process: first, determinize the NFA, and second, minimize the resulting DFA. To complete the first step, we use the *subset construction*, which I present briefly here. Let $A = (\Sigma, S, S^0, \rho, F)$ be an NFA. We construct $A_d = (\Sigma, 2^S, \{S^0\}, \rho_d, F_d)$, where $F_d = \{T \in 2^S : T \cap F \neq \emptyset\}$ and $\rho_d(T_1, a, T_2) \iff T_2 = \{t_2 \in S : \rho(t_1, a, t_2)$ for some $t_1 \in T_1\}$. The subset construction can be applied on the fly: starting with the initial state S^0, we determine the "next" state for each letter, and then recur. The automaton A_d is deterministic and accepts exactly the same language as A. For the second step, Watson [Wat93] presents 15 algorithms that can be used to minimize a DFA, including one of the simplest (Huffman's [Huf64]), and the one with the best known worst-case complexity (Hopcroft's [Hop71]). The second approach to canonization, due to Brzozowski [Brz62], avoids the minimization step, but applies the determinization step twice. In our study I compare the two approaches by evaluating the performance of Hopcroft's and Brzozowski's algorithms on randomly generated automata.

I present briefly the idea of the two algorithms. Let $L(A^{(p)})$ be the language accepted by the automaton A starting from the state p. Given a DFA, Huffman's and Hopcroft's algorithms construct an equivalence relation $E \subseteq S \times S$ with the following property: $(p, q) \in E \Leftrightarrow L(A^{(p)}) = L(A^{(q)})$. The equivalence relation E is computed as the greatest fixed point of the equation

$$(p, q) \in E \Leftrightarrow (p \in F \Leftrightarrow q \in F) \wedge (\forall a \in \Sigma, (p, a, p') \in \rho, (q, a, q') \in \rho : (p', q') \in E).$$

In Huffman's algorithm all states are assumed equivalent until proven otherwise. Equivalence classes are split repeatedly until a fix-point is reached. The algorithm runs in asymptotic time $O(|S|^2)$. Hopcroft made several clever optimizations in the way equivalence classes are split, which allowed him to achieve the lowest known running time $O(|S| \log |S|)$ [Gri73, Hop71]. Hopcroft's algorithm also significantly outperforms Huffman's algorithm in practice, so I can ignore Huffman's algorithm from this point on. Strictly speaking, Hopcroft's algorithm is just the DFA minimization algorithm, but I take it here to refer to the canonization algorithm, with determinization in the first step and minimization in the second step. Because the subset construction is applied in the first step, the worst-case complexity of this approach is exponential.

Brzozowski's algorithm is a direct canonization algorithm, and it does not use minimization, but, rather, two determinization steps. To describe the algorithm, I introduce some notation. If A is an automaton $(\Sigma, S, S^0, \rho, F)$, then $reverse(A)$ is the automaton $A^R = (\Sigma, S, F, \rho^R, S^0)$, where $\rho^R \subseteq S \times \Sigma \times S$ and $(s_2, a, s_1) \in \rho^R \Leftrightarrow (s_1, a, s_2) \in \rho$. Intuitively, *reverse* switches the accepting and the initial states, and changes the direction of the transitions. Let $determinize(A)$ be the deterministic automaton obtained from A using the subset construction, and let $reachable(A)$ be the automaton A with all states not reachable from the initial states removed.

Theorem (Brzozowski). *Let A be an NFA. Then*

$$A' = [reachable \circ determinize \circ reverse]^2(A)$$

is the minimal DFA accepting the same language as A.

It is not immediately obvious what the complexity of Brzozowski's algorithm is. The key to the correctness of the algorithm is, however, the following lemma.

Lemma. *Let $A = (\Sigma, S, \{s_0\}, \rho, F)$ be a DFA with the property that all states in S are reachable from s_0. Then $reachable(determinize(reverse(A)))$ is a minimal-state DFA.*

Clearly the resulting automaton A' is deterministic. Since we reverse twice, the languages accepted by A and A' are the same. The only thing that remains to be shown is that A' is minimal.

Theorem. *Let $A = (\Sigma, S, \rho, \{s_0\}, F)$ be a DFA with the property that all states in S are reachable from s_0. Then $A'' = reachable(determinize(reverse(A)))$ is a minimal-state DFA.*

Proof. Define $A^{(q)}$ to be the automaton $(\Sigma, S, \rho, \{q\}, F)$, that is, $A^{(q)}$ is the automaton A with starting state q. Also define $p \approx_A q \Leftrightarrow L(A^{(q)}) = L(A^{(p)})$. For brevity I use A^R to denote the automaton $reverse(A)$.

Now, to show that $A'' = (\Sigma, S'', \rho'', \{s_0''\}, F'')$ is minimal, we have to show that $\forall p, q \in S'', p \approx_{A''} q \Leftrightarrow p = q$. By construction the states of A'' are subsets of S, so in particular $p, q \subseteq S$. I will show that p and q are the same set iff they define the same language.

Let $r \in p \subseteq S$. Then there must be some word $w \in \Sigma^*$ such that $\rho^*(s_0, w) = r$, due to the assumption that every state in A is reachable from s_0. But then $s_0 \in (\rho^R)^*(r, w^R)$ in A, which means that $w^R \in L\left((A^R)^{(r)}\right)$. By construction, A'' is $determinize(A^R)$, therefore $w^R \in L(A''^{(p)})$. Our assumption was that $p \approx_{A''} q$, so it must also hold that $w^R \in L(A''^{(q)})$.

We take some state $t \in q \subseteq S$ such that $\rho^*(s_0, w) = t$. But ρ is deterministic, therefore $t = r$ and therefore $r \in q$. This proves that $p \subseteq q$. The proof that $q \subseteq p$ is analogous. Therefore $p = q$.

$\square$

Since the canonical size is at most exponential in the size of the input automaton and since *reachable* and *determinize* can be combined to generate only reachable sets

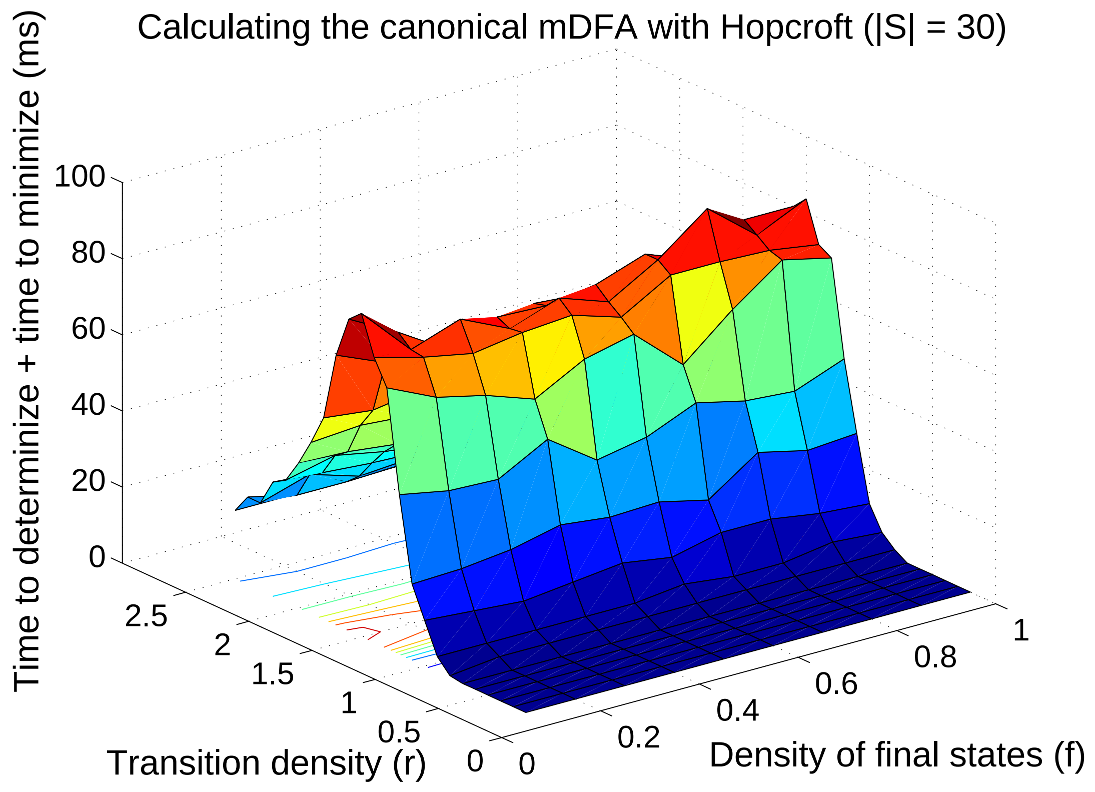


Figure 4.1: Canonization using Hopcroft

(which is exactly what we do in Hopcroft's algorithm), it follows that the worst-case complexity of Brzozowski's algorithm is also exponential.

4.3 Evaluating the Minimization Algorithms

For the experimental study I used the tool dk.brics.automaton [Mø04], developed by Anders Møller. I first study the performance on fixed-size automata. Again, the sample contains 200 random automata per (r, f) pair, and median time is reported (median time is less affected by outlying points than the mean. These, and all subsequent timing data, refer to the median). To generate each data point in Figure 4.1,

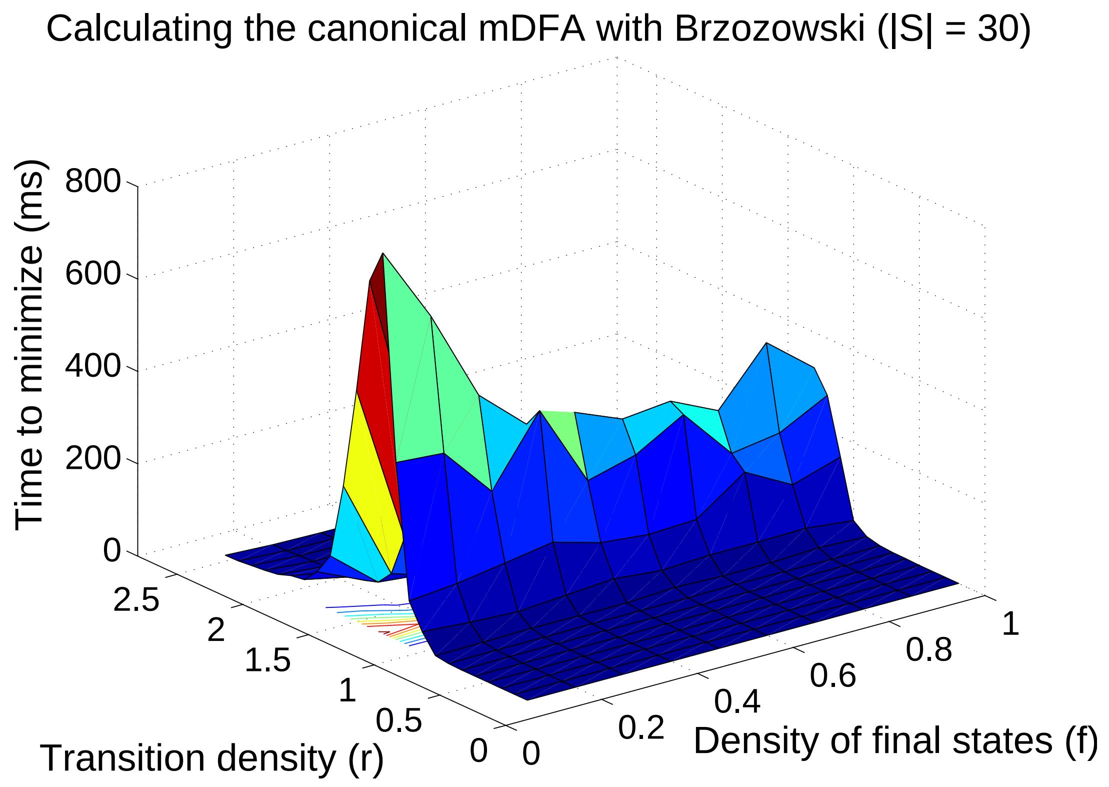


Figure 4.2: Canonization using Brzozowski

I determinized and then minimized with Hopcroft's algorithm each automaton; I measured combined running time for both steps. Note that Figure 4.1 is similar to Figure 2.4, but the two peaks occur in different densities ($r = 1.5$ and $r = 1.25$, respectively). As in Figure 2.4, for a fixed transition density, the impact of acceptance density on running time is not large.

For Brzozowski's algorithm, I measured the total time to perform the two $reachable \circ determinize \circ reverse$ steps. The results are presented in Figure 4.2. The peak for Brzozowski's algorithm coincides with the peak of Figure 2.4 ($r = 1.25$). For a fixed transition density, the impact of acceptance density on running time is much more pronounced that in Hopcroft's algorithm.

The experimental results indicate that neither Hopcroft's nor Brzozowski's algorithm dominates the other across the whole density landscape. Figures 4.3 and 4.3 show the running times of both algorithms for fixed $f = 0.5$. In Figure 4.3 the areas under both curves are 691 for Hopcroft and 995.5 for Brzozowski, and in Figure 4.3 the areas are 1900 for Hopcroft and 5866 for Brzozowski, so Hopcroft's algorithm has a better overall performance, but for $r > 1.5$ Brzozowski's algorithm is consistently faster. The conclusion is that Hopcroft's algorithm is faster for low-density automata, while Brzozowski's algorithm is better for high-density automata. It remains to be seen if this conclusion applies also for automata that arise in practical applications, e.g, [ABG$^+$00].

Similar to the approach of [PV04], I also investigated how Hopcroft's and Brzozowski's algorithms scale with automaton size. I fixed the acceptance density at $f = 0.5$, because its effect on the running time is less dramatic than that of the transition density. The results (Figures 4.5 and 4.6) indicate that none of the algorithms scales better than the other over the whole landscape. Brzozowski's algorithm has an edge over Hopcroft's for $r \geq 1.5$, and the opposite is true for the lower densities. At the peak, Hopcroft's algorithm scales exponentially, but generally the algorithms scale subexponentially. Again we see that Hopcroft's algorithm is better at low densities, while Brzozowski's algorithm is better at high densities.

The empirical measurements presented here suggest that none of the leading two minimization algorithms dominates the other for all transition densities. If one does

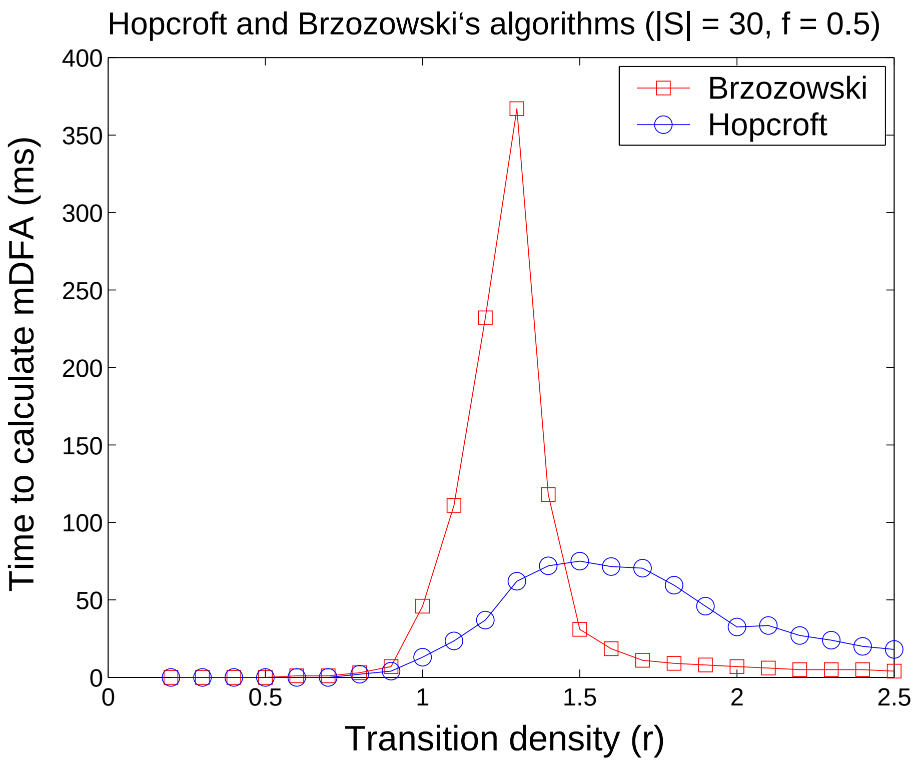


Figure 4.3: Comparison between Hopcroft's and Brzozowski's algorithms for fixed $f = 0.5$, using automata with size 30

not know a priori the density of the automaton it is better to choose Hopcroft's algorithm because its performance shows less variance. On the other hand, if the density is known and is above the cross-over threshold (which is found experimentally to be around density $r = 1.5$) choosing Brzozowski's algorithm might lead to performance gain over Hopcroft's algorithm.

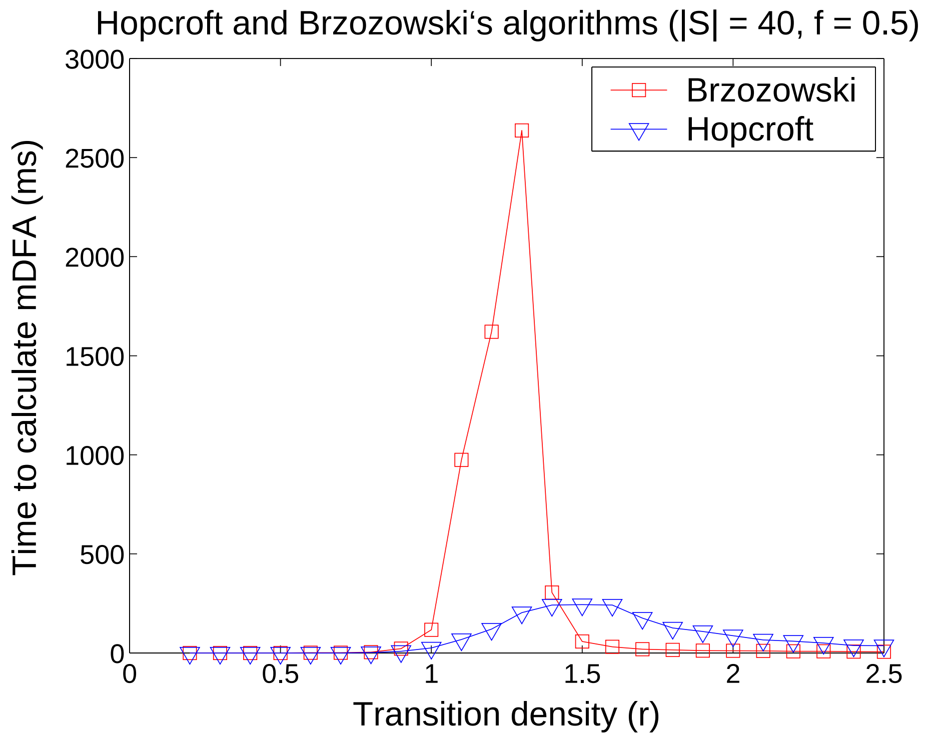


Figure 4.4: Comparison between Hopcroft's and Brzozowski's algorithms for fixed $f = 0.5$, using automata with size 40

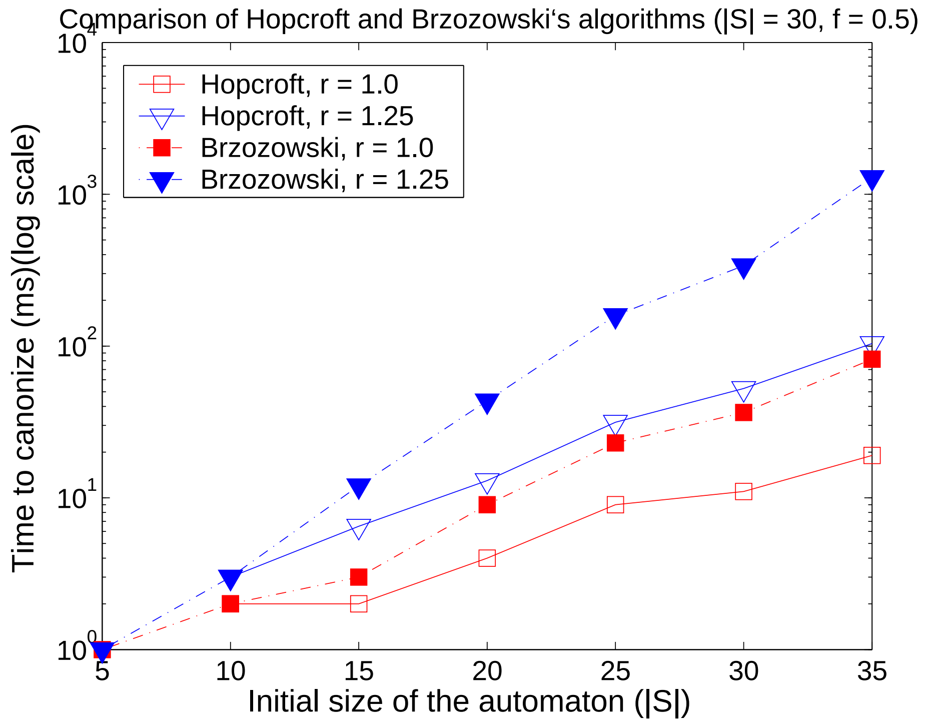


Figure 4.5: Scaling comparison of Hopcroft and Brzozowski's algorithms: low density

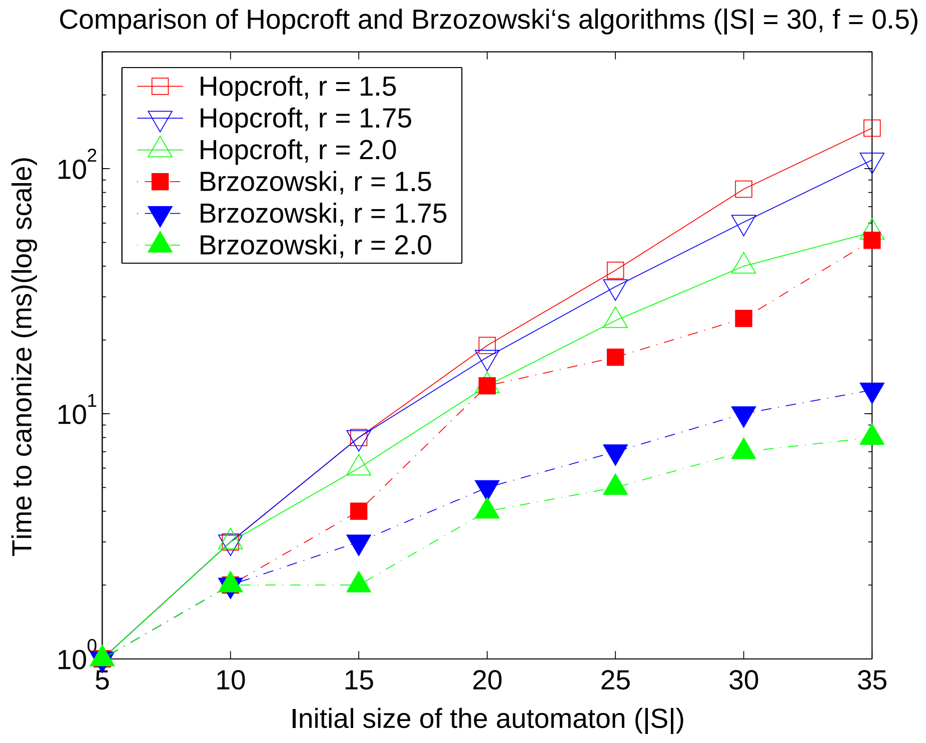


Figure 4.6: Scaling comparison of Hopcroft and Brzozowski's algorithms: high density

Chapter 5

Conclusions

This work proposed a probabilistic benchmark for testing automata-theoretic algorithms. I showed that in this model Hopcroft's and Brzozowski's canonization algorithms are incomparable, each having an advantage in a certain region of the model. In contrast, the advantage of the explicit approach to universality over the symbolic approach is quite clear.

An obvious question to raise is how "realistic" our probabilistic model is. There is no obvious answer to this question; partly because we lack realistic benchmarks of finite automata. Since automata represent finite-state control, it is hard to see why random directed graphs with linear density do not provide a realistic model. Hopefully, with the recent increase in popularity of finite-state formalisms in industrial temporal property specification languages (c.f., [AFF$^+$02, BBE$^+$01]), such benchmarks will become available in the not-too-far future, enabling me to evaluate the findings in this work on such benchmarks. While the results presented here are purely empirical, as the lack of success with fully analyzing related probabilistic models indicates (cf. [Fri99, DBM00, Ach00]), providing rigorous proof for these qualitative observations may be a very challenging task. At any rate, gaining a deeper understanding why one method is better than another method is an important challenge. Another research direction is to consider minimization on the fly, as, for example, in [LY92].

The most surprising result of this work is the superiority of the explicit approach to universality over the symbolic approach. This runs against the conventional wisdom in verification [BCM$^+$92]. One may wonder whether the reason for this is the fact that the sequential circuits derived from the random model can be viewed as consisting of "pure control", with no data component, unlike typical hardware designs, which combine control and data. This suggests that perhaps in model checking such designs, control and data ought to be handled by different techniques.

In future work I will extend the comparison between the explicit and symbolic approaches to universality to automata on infinite words, a problem of very direct relevance to computer-aided verification [KV01b]. It is known that complementation of such automata is quite intricate [KV01b], challenging both explicit and symbolic implementation.

Appendix A
Encoding the Universality Problem into Promela: an Example

Here I present the full Promela encoding of the automaton on Figure A.1.

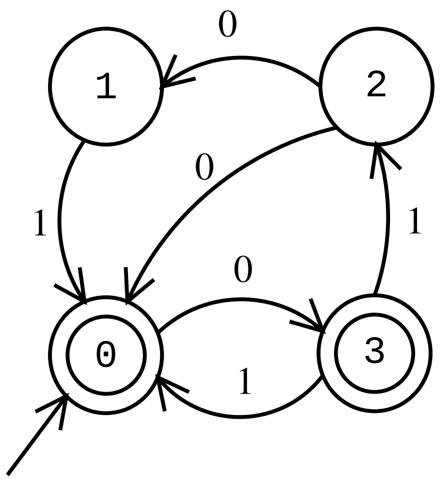


Figure A.1: Example Automaton

```
bool letterA;

active proctype model() {
  bool state[4] = false;
  bool nextstate[4] = false;
  state[0] = true;

  int i;
  do
    ::
      atomic {
      /* Reset everything in nextstate[] to false */
      i = 0;
      do
        :: (i < 4) ->
              nextstate[i] = false;
              i++;
        :: else -> break;
      od;

      if
        :: (state[0] && letterA) ->
              skip;
        :: (state[0] && ! letterA) ->
              nextstate[3] = true;
        :: else -> skip;
      fi;

      if
        :: (state[1] && letterA) ->
              nextstate[0] = true;
        :: (state[1] && ! letterA) ->
              skip;
        :: else -> skip;
      fi;

      if
        :: (state[2] && letterA) ->
              skip;
        :: (state[2] && ! letterA) ->
              nextstate[0] = true;
              nextstate[1] = true;
        :: else -> skip;
      fi;

      if
        :: (state[3] && letterA) ->
              nextstate[0] = true;
              nextstate[2] = true;
        :: (state[3] && ! letterA) ->
              skip;
        :: else -> skip;
      fi;

      /* Make the transition to the next state */
      i=0;
      do
```

```
        :: (i < 4) ->
              state[i] = nextstate[i];
              i++;
        :: else -> break;
      od;

      assert (  state[0] || state[3] );
    }
  od;
}


active proctype driver() {
  do
    :: letterA = true;
    :: letterA = false;
  od;
}
```

Bibliography

[ABG$^+$00] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfstal. FoCs - automatic generation of simulation checkers from formal specifications. In *CAV, Proc. 12th International Conference*, volume 1855 of *LNCS*, pages 538–542. Springer-Verlag, 2000.

[Ach00] D. Achlioptas. Setting two variables at a time yields a new lower bound for random 3-SAT. In *Proc. of 32nd Annual ACM Symposium on Theory of Computing*, 2000.

[AFF$^+$02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.

[AV01] A. San Miguel Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In *Principles and Practice of Constraint Programming*, pages 121–136, 2001.

[BB94] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Design Automation Conference*, pages 596–602. IEEE Computer Society, 1994.

[BBE$^+$01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Proc. 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 363–367, Paris, France, July 2001. Springer-Verlag.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proc. IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, pages 49–58, 1991.

[BCM$^+$92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Bol01] B. Bollobas. *Random Graphs.* Cambridge University Press, January 2001.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Brz62] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962. Volume 12 of MRI Symposia Series.

[Cad] Cadence. SMV. http://www.cadence.com/company/cadence_labs_research.html.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *IJCAI '91*, pages 331–337, 1991.

[DBM00] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *SODA*, pages 126–127, 2000.

[Fri99] E. Friedgut. Necessary and sufficient conditions for sharp thresholds of graph properties, and the k-SAT problem. *Journal of the A.M.S.*, 12:1017–1054, 1999.

[GG97] James Glenn and William I. Gasarch. Implementing WS1S via finite automata: Performance issues. In *Workshop on Implementing Automata*, pages 75–86, 1997.

[Gri73] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[Hol04] G. J. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HS96] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Huf64] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers.* Addison-Wesley, 1964.

[Kar90] R. M. Karp. The transitive closure of a random digraph. *Random Struct. Algorithms*, 1(1):73–94, 1990.

[KBvH01] A. M. C. A. Koster, H. L. Bodlaender, and C. P. M. van Hoesel. Treewidth: Computational experiments. ZIB-Report 01–38, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Berlin, Germany, 2001. Also available as technical report UU-CS-2001-49 (Utrecht University) and research memorandum 02/001 (Universiteit Maastricht).

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, Boston, MA., 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.

[KV99] O. Kupferman and M.Y. Vardi. Model checking of safety properties. In *Computer Aided Verification, Proc. 11th International Conference*, volume 1633 of *LNCS*, pages 172–183. Springer-Verlag, 1999.

[KV01a] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3):291–314, November 2001.

[KV01b] O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. on Computational Logic*, 2001(2):408–429, July 2001.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 264–274, Victoria, May 1992.

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

[MF71] A.R. Meyer and M.J. Fischer. Economy of description by automata, grammars, and formal systems. In *Proc. 12th IEEE Symp. on Switching and Automata Theory*, pages 188–191, 1971.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[Mø04] A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.

[Moo03] J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[PV04] G. Pan and M.Y. Vardi. Search vs. symbolic techniques in satisfiability solving. In *SAT 2004*, LNCS, Aalborg, May 2004. Springer-Verlag.

[Rus98] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/user/russ/david/k7-div-sqrt.html.

[SML96] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[SRA03] Koushik Sen, Grigore Roşu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In Vijay A. Saraswat, editor, *ASIAN*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2003.

[TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

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

[Wat93] B. W. Watson. A taxonomy of finite automata minimization algorithmes. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.