Mona: Monadic Second-Order Logic in Practice

Jesper G. Henriksen[*]
Jakob Jensen[*]
Michael Jørgensen[*]
Nils Klarlund[†]
Robert Paige[‡]
Theis Rauhe[*]
Anders Sandholm[*]

ABSTRACT [1] The purpose of this article is to introduce Monadic Second-order Logic as a practical means of specifying regularity. The logic is a highly succinct alternative to the use of regular expressions. We have built a tool MONA, which acts as a decision procedure and as a translator to finite-state automata. The tool is based on new algorithms for minimizing finite-state automata that use binary decision diagrams (BDDs) to represent transition functions in compressed form. A byproduct of this work is an algorithm that matches the time but improves the space of Sieling and Wegener's algorithm to reduce OBDDs in linear time.

The potential applications are numerous. We discuss text processing, Boolean circuits, and distributed systems. Our main example is an automatic proof of properties for the "Dining Philosophers with Encyclopedia" example by Kurshan and MacMillan. We establish these properties for the parameterized case *without* the use of induction.

Our results show that, contrary to common beliefs, high computational complexity may be a desired feature of a specification formalism.

---

[*]BRICS, Centre of the Danish National Research Foundation for Basic Research in Computer Science, Department of Computer Science, University of Aarhus.

[†]The corresponding author is Nils Klarlund, who is with BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C. E-mail: klarlund@daimi.aau.dk.

[‡]Department of Computer Science, CIMS, New York University, 251 Mercer St. New York, New York, USA; research partially supported by ONR grant N00014-93-1-0924, AFOSR grant AFOSR-91-0308, and NSF grant MIP-9300210.

[1]This article is a heavily revised version of [JJK94].

1 Introduction

In computer science, *regularity* amounts to the concept that a class of structures is recognized by a finite-state device. Often phenomena are so complicated that their regularity either

- may be overlooked, as in the case of parameterized verification of distributed finite-state systems with a regular communication topology; or
- may not be exploited, as in the case when a search pattern in a text editor is known to be regular, but in practice inexpressible as a regular expression.

In this paper we argue that the *Monadic Second-Order Logic* or *M2L* can help in practice to identify and to use regularity. In M2L, one can directly mention positions and subsets of positions in the string. This feature distinguishes the logic from regular expressions or automata. Together with quantification and Boolean connectives, an extraordinary succinct formalism arises.

Although it has been known for thirty-five years that M2L defines regular languages (see [Tho90]), the translator from formulas to automata that we describe in this article appears to be one of the first implementations.

The reason such projects have not been pursued may be the staggering theoretical lower-bound: any decision procedure is bound to sometimes require as much time as a stack of exponentials that has height proportional to the length of the formula.

It is often believed that the lower the computational complexity of a formalism is, the more useful it may be in practice. We want to counter such beliefs in this article — at least for logics on finite strings.

1.1 Why use logic?

Some simple finite-state languages easily described in English call for convoluted regular expressions. For example, the language L_{2a2b} of all strings over $\Sigma = \{a, b, c\}$ containing at least two occurrences of a *and* at least two occurrences of b seems to require a voluminous expression, such as

$$\begin{array}{ll} & \Sigma^* a \Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* \\ \cup & \Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* \\ \cup & \Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* b \Sigma^* . \end{array}$$

If we added $\cap$ to the operators for forming regular expressions, then the language L_{2a2b} could be expressed more concisely as $(\Sigma^* a \Sigma^* a \Sigma^*) \cap (\Sigma^* b \Sigma^* b \Sigma^*)$.

Even with this extended set of operators, it is often more convenient to express regular languages in terms of positions and corresponding letters. For example, to express the set $L_{a\text{after}b}$ of strings in which every b is followed by an a, we would like a formal language allowing us to write something like

> "for every position p, if there is a b in p then for some position q after p, there is an a in q."

The extended regular languages do not seem to allow an expression that very closely reflects this description — although upon some reflection a small regular expression can be found. But in M2L we can express $L_{a\text{after}b}$ by a formula

$$\forall p : {'b'}(p) \;\Rightarrow\; \exists q :\; p < q \;\wedge\; {'a'}(q)$$

(Here the predicate ${'b'}(p)$ means "there is a b in position p".) In general, we believe that many errors can be avoided if logic is used when the description in English does not lend itself to a direct translation into regular expressions or automata. However, the logic can easily be combined with other methods of specifying regularity since almost any such formalism can be translated with only a linear blow-up into M2L.

Often regularity is identified by means of *projections*. For example, if L_{trans} is regular on a cross-product alphabet $\Sigma \times \Sigma$ (e.g. describing a parameterized transition relation, see Section 5) and L_{start} is a regular language on Σ describing a set of start strings, then the set of strings that can be reached by a transition from a start string is $\pi_2(L_{trans} \cap \pi_1^{-1}(L_{start}))$, where π_1 and π_2 are the projections from $(\Sigma \times \Sigma)^*$ to the first and second component. Such language-theoretic operations can be very elegantly expressed in M2L.

1.2 Our results

In this article, we discuss applications of M2L to text processesing and the description of parameterized Boolean circuits. Our principal application is a new proof technique for establishing properties about parameterized, distributed finite-state systems with regular communication topology. We illustrate our method by showing safety and liveness properties for a non-trivial version of the Dining Philosophers' problem as proposed in [KM89] by Kurshan and MacMillan.

We present MONA, which is our tool that translates formulas in M2L to finite-state machines. We show how BDDs can be used to overcome an otherwise inherent problem of exponential explosion. Our minimization algorithm works very fast in practice thanks to a simple generalization of the unary apply operation of BDDs.

1.3 Comparisons to other work

Parameterized circuits are described using BDDs in [GF93]. This method relies on formulating inductive steps as finite-state devices and does not provide a single specification language. The work in [RS93] is closer in spirit to our method in that languages of finite strings are used although not as part of a logical framework. In [BSV93], another approach is given based on iterating abstractions. The parameterized Dining Philosopher's problem is solved in [KM89] by a finite-state induction principle.

A tool for M2L on finite, binary trees has been developed at the University of Kiel [Ste93]. Apparently, this tool has only been used for very simple examples.

In [CR94], a programming language for finite domains based on a fixed point logic is described and used for verification of non-parameterized finite systems.

1.4 Contents

In Section 2, we explain the syntax and semantics of M2L on strings. We recall the correspondence to automata theory in Section 3. We give several applications of M2L and the tool in Section 4: text patterns, parameterized circuits, and equivalence testing. Our main example of parameterized verification is discussed in Section 5. We give an overview of our implementation in Section 6.

2 The Monadic Second-order Logic on Strings

Let Σ be an alphabet and let w be a string over Σ. The semantics of the logic determines whether a closed M2L formula ϕ holds on w. The language $L(\phi)$ denoted by ϕ is the set of strings that make ϕ hold. Assume now that w has length n and consists of letters $a_0a_1...a_{n-1}$. The *positions* in w are then $0,...,n-1$. We can now describe the three syntactic categories of M2L on strings.

A *position term* t is either

- the constant 0 (which denotes the position 0);
- the constant \$ (which denotes the last position, i.e. $n-1$);
- a position variable p (which denotes a position i);
- of the form $t \oplus i$ (which denotes the position $j + i \bmod n$, where j is the interpretation of t); or
- of the form $t \ominus i$ (which denotes the position $j - i \bmod n$, where j is the interpretation of t);

(Position terms are only interpreted for non-empty strings).
A *position set term* T is either

- the constant $\emptyset$ (which denotes the empty set);
- the constant **all** (which denotes the set $\{0, ..., n-1\}$);
- a position set variable P (which denotes a subset of positions);
- of the form $T_1 \cup T_2$, $T_1 \cap T_2$, or $\complement T_1$ (which are interpreted in the natural way);
- of the form $T+i$ (which denotes the set of positions in T shifted right by an amount of i); or
- of the form $T-i$ (which denotes the set of positions in T shifted left by an amount of i);

A *formula* ϕ is either of the form

- $'a'(t)$ (which holds if letter a_i in $w = a_0 a_1 \cdots$ is a, where i is the interpretation of t);
- $t_1 = t_2$, $t_1 < t_2$ or $t_1 \leq t_2$ (which are interpreted in the natural way);
- $T_1 = T_2$, $T_1 \subseteq T_2$, or $t \in T$ (which are interpreted in the natural way);
- $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \Rightarrow \phi_2$, or $\phi_1 \Leftrightarrow \phi_2$ (where ϕ_1 and ϕ_2 are formulas, and which are interpreted in the natural way);
- $\exists p : \phi$ (which is true, if there is a position i such that ϕ holds when i is substituted for p);
- $\forall p : \phi$ (which is true, if for all positions i, ϕ holds when i is substituted for p);
- $\exists P : \phi$ (which is true, if there is a subset of positions I such that ϕ holds when I is substituted for P); or
- $\forall P : \phi$ (which is true, if for all subsets of positions I, ϕ holds when I is substituted for P);

3 From M2L to Automata

In this section, we recall the method for translating a formula in M2L to an equivalent finite-state automaton (see [Tho90] for more details). Note that any formula ϕ can be interpreted, given a string w and a *value assignment* $\mathcal{I}$ that fixes values of the free variables. If ϕ then holds, we write $w, \mathcal{I} \models \phi$. The key idea is that a value assignment and the string may be described together as a word over an extended alphabet consisting of Σ and extra binary tracks, one for each variable. By structural induction, we then define for each formula an automaton that exactly recognizes the words in the extended alphabet corresponding to pairs consisting of a string and an assignment that satisfy the formula.

Example

Assume that the free variables are $\mathcal{P} = \{P_1, P_2\}$ and that $\Sigma = \{a, b\}$. Let us consider the string $w = abaa$ and value assignment

$$\mathcal{I} = [P_1 \mapsto \{0,2\}, P_2 \mapsto \emptyset].$$

The set $\mathcal{I}(P_1) = \{0, 2\}$ can be represented by the bit pattern 1010, since the numbered sequence

$$\begin{array}{cccc} 1 & 0 & 1 & 0 \\ {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} \end{array}$$

defines that 0 is in the set (the bit in position 0 is 1), 1 is not in the set (the bit in position 1 is 0), etc. Similarly, the bit pattern 0000 describes $\mathcal{I}(P_2) = \emptyset$.

If these patterns are laid down as extra "tracks" along w, we obtain an *extended word* α, which may be depicted as:

a	b	a	a
1	0	1	0
0	0	0	0

Technically, we define $\alpha = \alpha_0 \cdots \alpha_3$ as the word $(a, 1, 0)(b, 0, 0)(a, 1, 0)$ $(a, 0, 0)$ over the alphabet $\Sigma \times \mathbb{B} \times \mathbb{B}$ of *extended letters*, where $\mathbb{B} = \{0, 1\}$ is the set of truth values.

This correspondence can be generalized to any w and any value assignment for a set of variables $\mathcal{P}$ (which can all be assumed to be second-order).

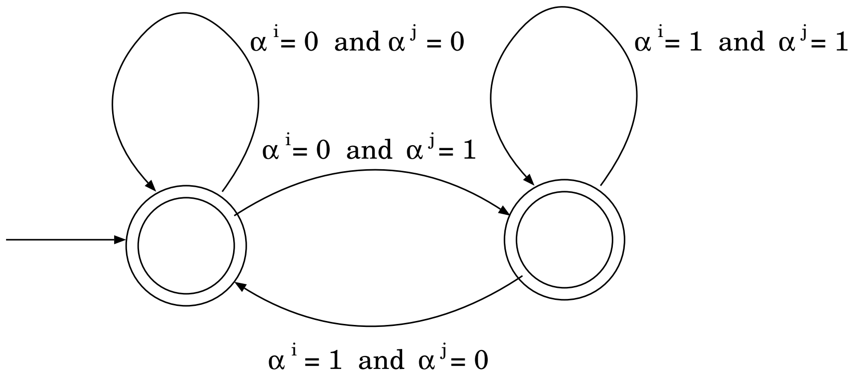
By structural induction on formulas, we construct automata $A^{\phi,\mathcal{P}}$ over alphabet $\Sigma \times \mathbb{B}^k$—where $\mathcal{P} = \{P_1, \cdots, P_k\}$ is any set of variables containing the free variables in ϕ—satisfying the *fundamental correspondence*:

$$w, \mathcal{I} \models \phi \text{ iff } (w, \mathcal{I}) \in L(A^{\phi,\mathcal{P}})$$

Thus $A^{\phi,\mathcal{P}}$ accepts exactly the pairs $(w, \mathcal{I})$ that make ϕ true.

Example

Let ϕ be the formula $P_i = P_j + 1$. Thus when ϕ holds, P_i is represented by the same bit pattern as that of P_j but shifted right by one position. This can be expressed by the automaton $A^{\phi,\mathcal{P}}$:



In this drawing, α^i refers to the ith extra track. Thus, the automaton checks that the ith track holds the same bit as the jth track the instant before.

4 Applications

4.1 Text patterns

The language L_{2a2b} of strings containing at least two occurrences of a and two occurrences of b can be described in M2L by the formula

$$\begin{array}{l}(\exists p_1, p_2 : {'a'}(p_1) \ \wedge \ {'a'}(p_2) \ \wedge \ p_1 \neq p_2) \ \wedge \\ (\exists p_1, p_2 : {'b'}(p_1) \ \wedge \ {'b'}(p_2) \ \wedge \ p_1 \neq p_2)\end{array}$$

Our translator yields the minimal automaton, which contains nine states, in a fraction of a second.

The language $L_{a\text{after}b}$ given by the formula

$$\forall p : {'b'}(p) \ \Rightarrow \ \exists q : \ p < q \ \wedge \ {'a'}(q)$$

is translated to the minimal automaton, which has two states, in .3 seconds.

A far more complicated language to express is $L_{<1\text{apart}}$ consisting of every string over $\{a, b\}$ such that for any prefix the number of a's and b's are at most one apart. When using regular expressions or M2L, one needs to struggle a bit, but in M2L there is a strategy for describing the functioning of the finite-state machine that comes to mind.

We observe that a position p may be used to designate a prefix; for example, 0 denotes the prefix consisting of the first letter and \$ (the last position) denotes the whole input string. We may now recognize a string in $L_{<1\text{apart}}$ by identifying three sets of positions: the set P_0 corresponding to prefixes with an equal number of a's and b's, the set P_{+1} corresponding to prefixes where the number of a's is one greater than the number of b's,

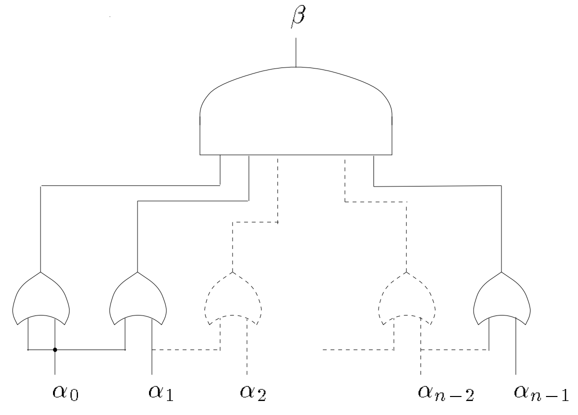


FIGURE 1. A parameterized circuit.

and the set P_{-1} corresponding to prefixes where the number of a's is one less than the number of b's:

$$\begin{aligned}
\exists P_0, P_{+1}, P_{-1} : P_0 \cup P_{+1} \cup P_{-1} = \mathbf{all} \\
\wedge\ 0 \notin P_0 \\
\wedge\ 0 \in P_{+1} \Leftrightarrow {'a'}(0) \\
\wedge\ 0 \in P_{-1} \Leftrightarrow {'b'}(0) \\
\wedge\ \forall p : (p > 0 \ \Rightarrow \\
p \in P_0 \ \Leftrightarrow\ ({'a'}(p)\ \wedge\ p \ominus 1 \in P_{-1}) \\
\vee\ ({'b'}(p)\ \wedge\ p \ominus 1 \in P_{+1}) \\
\wedge\ p \in P_{+1}\ \Leftrightarrow\ {'a'}(p)\ \wedge\ p \ominus 1 \in P_0 \\
\wedge\ p \in P_{-1}\ \Leftrightarrow\ {'b'}(p)\ \wedge\ p \ominus 1 \in P_0)
\end{aligned}$$

The resulting four-state automaton is calculated in a fraction of a second.

4.2 Parameterized circuits

Assume that we are given a drawing as in Figure 1 denoting a parameterized Boolean function.

How do we describe the language $L_{ex} \subseteq \mathbb{B}^*$ of input bit patterns that make the output true? From the drawing, no immediate description as a regular expression or finite-state automaton is apparent. In M2L, however, it is easy to model the outputs of the n or-gates as a second-order variable Q, which allows the language to be described from a direct interpretation of the drawing. Note that the or-gate at position $p > 0$ is true if either there is a 1 at $p - 1$ or p, or in other words: $p \in Q \Leftrightarrow {'1'}(p \ominus 1) \vee {'1'}(p)$. Since the output is 1 if and only if all or-gates are 1, i.e. if $Q = \mathbf{all}$, the language L_{ex} is given by the formula

$$\begin{aligned}
\exists Q : (\forall p : \quad & (p = 0 \Rightarrow p \in Q \Leftrightarrow {'1'}(p))\ \wedge \\
& (p > 0 \Rightarrow (p \in Q \Leftrightarrow {'1'}(p \ominus 1) \vee {'1'}(p)))\ \wedge Q = \mathbf{all})
\end{aligned}$$

The resulting automaton has three states and accepts the language $(1 \cup 10)^*$, which is the regular expression that one would obtain by reasoning about the circuit. For more advanced applications to hardware verification, see [BK95].

4.3 Equivalence testing

A closed formula ϕ is a *tautology* if $L(\phi) = L(\Sigma^*)$, i.e. if all strings over Σ satisfy ϕ. The equivalence of formulas ϕ and ψ then amounts to whether $\phi \Leftrightarrow \psi$ is a tautology.
Example. That a set P contains exactly the even positions in a non-empty input string may be expressed in M2L by the following two rather different approaches: either by the formula *even1*$(P) \equiv$

$$\begin{aligned} 0 \in P \ \wedge \ \forall p : ((p \in P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \notin P) \\ \wedge \ (p \notin P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \in P)), \end{aligned}$$

or as a formula *even2*$(P) \equiv$

$$P \cup (P+1) = \mathbf{all} \ \wedge \ P \cap (P+1) = \emptyset \ \wedge \ P \neq \emptyset$$

To show the equivalence of the two formulas, we check the truth value of the bi-implication:

$$\forall P : even1(P) \Leftrightarrow even2(P)$$

The translation of this formula does indeed produce an automaton accepting Σ^*, and thus verifies our claim.

5 Dining Philosophers with Encyclopedia

A distributed system is *parameterized* when the number n of processes is not fixed a priori. For such systems the state space is unbounded, and thus traditional finite-state verification methods cannot be used. Instead, one often fixes n to be, say two or three. This yields a finite state space amenable to state exploration methods. However, the validity of a property for $n = 2, 3$ does not necessarily imply that the property holds for all n.

A central problem in verification is automatically to validate parameterized systems. One way to attack the problem is to formulate induction principles such that the base case and the inductive steps can be formulated as finite-state problems. Kurshan and MacMillan [KM89] used such a method to verify safety and liveness properties of a non-trivial version of the Dining Philosophers example.

In this system, symmetry is broken by an encyclopedia that circulates among the philosophers. Thus each philosopher is in one of three states:

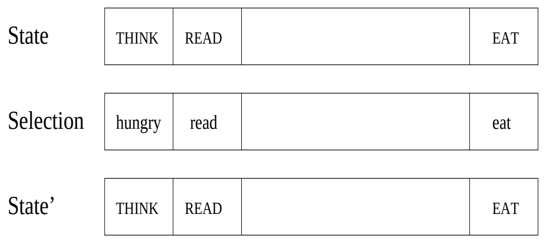


FIGURE 2. Dining Philosophers with Encyclopedia

EAT, THINK, or READ. The global state can be described as a string *State* of length n over the alphabet $\Sigma_{\text{State}} = \{\text{EAT}, \text{THINK}, \text{READ}\}$, see Figure 2.

The system makes a transition according to external events that constitute a *selection* . Each process is presented with an event in the alphabet $\Sigma_{\text{Selection}} = \{\text{eat}, \text{think}, \text{read}, \text{hungry}\}$. Thus the selection can be viewed as a string *Selection* over $\Sigma_{\text{Selection}}$, see Figure 2. As shown, all processes make a synchronous transition to a new global $State'$ on a selection according to a transition relation trans$(State, State', Selection)$, which is shown in Figure 3[2] together with an auxiliary predicate blocking$(Selection)$ used in its definition. Thus the new state of each process is dependent on its old state and on the selection events presented to itself and its neighbors. The transition relation is so complicated that it is hard to grasp the functioning of the system.

Fortunately, the parameterized transition relation can be translated into basic M2L on strings. For example, we encode *State* using two second-order variables P and Q with the convention that

$$\begin{array}{l}
\text{EAT}_p(State) \equiv p \in P \ \wedge \ p \in Q \\
\text{READ}_p(State) \equiv p \notin P \ \wedge \ p \in Q \\
\text{THINK}_p(State) \equiv p \notin P \ \wedge \ p \notin Q
\end{array}$$

Similarly, $State'$ and *Selection* can also each be encoded using two second-order variables. Thus, the predicate trans$(State, State', Selection)$ becomes a formula with six free second-order variables.

For this distributed system there are two important properties to verify:

- *Safety Property*: The encyclopedia is neither lost nor replicated. Thus there is always exactly one process in state READ.
- *Liveness Property*: If no process remains in state EAT forever, then the encyclopedia is passed around over and over.

In [KM89] both properties are proved in terms of a complicated induction hypothesis. This hypothesis is itself a distributed system, where each

[2]We use '#' in the beginning of a line to indicate that this line is a comment.

process has four states. (The Liveness Property in [KM89] is technically different since it is modeled in terms of selections.)

Our strategy is fundamentally different. We cannot directly verify liveness properties. But we can easily verify properties about the transition relation in the parameterized case and *without* induction as follows.

Let ϕ be an M2L formula about the global state. For example, we might consider the property that if a philosopher eats, then his neighbors do not:

$$\phi_{\text{mutex}}(State) \equiv \forall p : \text{EAT}_p(State) \Rightarrow \neg\text{EAT}_{p\ominus 1}(State) \ \wedge\ \neg\text{EAT}_{p\oplus 1}(State)$$

A property given as a formula ϕ can be verified using the *invariance principle*:

$$\begin{aligned}&\forall State, State', Selection :\\ &\quad \phi(State) \ \wedge\ \text{trans}(State, State', Selection) \Rightarrow \phi(State'),\end{aligned}$$

which is also a formula in M2L. In this way, we have verified for the parameterized case that both ϕ_{mutex} and the Safety Property that exactly one philosopher reads, i.e. $\exists! p : \text{READ}_p(State)$, are invariant. MONA verifies such a formula in approximately 3 seconds on a Sparc 20.

Note that this method does not rely on a state space exploration (which is impossible since the state space is unbounded). Instead, it is based on the Invariance Principle: to show that a property holds for all reachable states, it is sufficient to show that it holds for the initial state and is preserved under any transition.

5.1 *Establishing the liveness property*

The Liveness Property can be expressed in Temporal Logic as

$$\Box(\text{READ}_{p\ominus 1} \ \Rightarrow\ \Diamond\text{READ}_p), \tag{1.1}$$

that is, it always holds that if philosopher $p \ominus 1$ reads, then eventually philosopher p reads. We must prove this property under the assumption that no philosopher eats forever:

$$\Box(\text{EAT}_p \ \Rightarrow\ \Diamond\neg\text{EAT}_p). \tag{1.2}$$

So assume that $\text{READ}_{p\ominus 1}$ holds. We must prove that $\Diamond\text{READ}_p$ holds. There are two cases as follows.

- Case EAT_p holds. By asssumption (1.2), there is an instant when $\text{EAT}_p \ \wedge\ \neg \bigcirc \text{EAT}_p$ holds. Thus if

  $$\text{READ}_{p\ominus 1} \ \wedge\ \text{EAT}_p \ \wedge\ \neg \bigcirc \text{EAT}_p \Rightarrow \bigcirc\text{READ}_p \tag{1.3}$$

  is a valid property of the transition system, $\Diamond\text{EAT}_p$ holds. In fact, we verified using MONA that (1.3) indeed holds.

$\text{blocking}(\mathit{Selection}) \equiv$
$\text{eat}_{p\oplus 1}(\mathit{Selection}) \ \vee\ \text{hungry}_{p\ominus 1}(\mathit{Selection})$
$\vee\ \text{eat}_{p\ominus 1}(\mathit{Selection})$

$\text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \equiv$
$\forall p :$

#THINK $\rightarrow$ THINK :
$(\text{THINK}_p(\mathit{State}) \ \wedge\ \text{THINK}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \ \wedge\ \neg(\text{read}_{p\ominus 1}(\mathit{Selection}))$
$\vee$
$\text{hungry}_p(\mathit{Selection}) \ \wedge\ \text{blocking}(\mathit{Selection}))$

$\wedge$
#THINK $\rightarrow$ EAT :
$(\text{THINK}_p(\mathit{State}) \ \wedge\ \text{EAT}_p(\mathit{State}') \Rightarrow$
$\text{hungry}_p(\mathit{Selection}) \ \wedge\ \neg(\text{blocking}(\mathit{Selection})))$

$\wedge$
#THINK $\rightarrow$ READ :
$(\text{THINK}_p(\mathit{State}) \ \wedge\ \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \ \wedge\ \text{read}_{p\ominus 1}(\mathit{Selection}))$

$\wedge$
#EAT $\rightarrow$ THINK :
$(\text{EAT}_p(\mathit{State}) \ \wedge\ \text{THINK}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \ \wedge\ \neg(\text{read}_{p\ominus 1}(\mathit{Selection})))$

$\wedge$
#EAT $\rightarrow$ EAT :
$(\text{EAT}_p(\mathit{State}) \ \wedge\ \text{EAT}_p(\mathit{State}') \Rightarrow$
$\text{eat}_p(\mathit{Selection}))$

$\wedge$
#EAT $\rightarrow$ READ :
$(\text{EAT}_p(\mathit{State}) \ \wedge\ \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{think}_p(\mathit{Selection}) \ \wedge\ \text{read}_{p\ominus 1}(\mathit{Selection}))$

$\wedge$
#READ $\rightarrow$ THINK :
$(\text{READ}_p(\mathit{State}) \ \wedge\ \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{read}_p(\mathit{Selection}) \ \wedge\ \text{think}_{p\oplus 1}(\mathit{Selection}))$

$\wedge$
#READ $\rightarrow$ EAT :
$(\text{READ}_p(\mathit{State}) \ \wedge\ \text{EAT}_p(\mathit{State}') \Rightarrow$
false)

$\wedge$
#READ $\rightarrow$ READ :
$(\text{READ}_p(\mathit{State}) \ \wedge\ \text{READ}_p(\mathit{State}') \Rightarrow$
$\text{read}_p(\mathit{Selection}) \ \wedge\ \neg(\text{think}_{p\oplus 1}(\mathit{Selection})))$

FIGURE 3. The transition relation

- Case $\neg\mathrm{EAT}_p$ holds. If EAT_p becomes true, then use the previous case. Otherwise, $\neg\mathrm{EAT}_p$ continues to hold. Now, by the assumption (1.2) at some point $\neg\mathrm{EAT}_{p\oplus 1}$ will hold. We then use the property

  $$\mathrm{READ}_{p\ominus 1} \wedge \neg\mathrm{EAT}_p \wedge \neg\bigcirc\mathrm{EAT}_{p\oplus 1} \Rightarrow \bigcirc\mathrm{READ}_p \vee \bigcirc\mathrm{EAT}_p, \quad (1.4)$$

  which we have also verified using MONA, to show that eventually READ_p holds (or eventually EAT_p holds, which contradicts the assumption that $\neg\mathrm{EAT}_p$ continues to hold).

6 Implementation.

MONA is our implementation of the decision procedure, which translates formulas of M2L to finite-state automata as outlined in Section 3. Our tool is implemented in Standard ML of New Jersey. A previous version of MONA was written in C with explicit garbage collection and based on representing transition functions in a conjunctive normal form. Our present tool runs up to 50 times faster due to improved algorithms.

6.1 Representation of automata

Since the size of the extended alphabet grows exponentially with the number of variables, a straightforward implementation based on explicitly representing the alphabet would only work for very simple examples. Instead, we represent the transition relation using Binary Decision Diagrams (B-DDs) [Bry92, Bry86]. In this way, the alphabet is never explicitly represented. For the external alphabet of ASCII-characters, we choose an encoding based on seven extra tracks holding the binary representation. Thus, character classes such as `[a-zA-Z]` become represented as very simple BDDs.

A deterministic automaton A is represented as follows. The state space is $Q = \{0, 1, \ldots, n-1\}$, where n is size of the state space; $\mathbb{B}^k$ is the extended alphabet; $i_0 \in Q$ is the initial state; $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function; and $F \subseteq Q$ is the set of accepting states. We use a bit vector of size n to represent F and an array containing n pointers to roots of multi-terminal BDDs representing δ. A leaf of a BDD holds the integer designating the next state. An internal node v is called a *decision node* and contains an *index* denoted $v.index$, where $0 \leq v.index < k$, and high and low successors $v.hi$ and $v.lo$. If b is a sequence of k bits, i.e. $b \in \mathbb{B}^k$, then $\delta(q, b)$ is found by looking up the qth entry in the array and following the decision nodes according to b until a leaf is reached (node v is followed by selecting the high successor if the $v.index$th component of b is 1 and the low successor if it is 0).

For example, the following finite automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter “11”

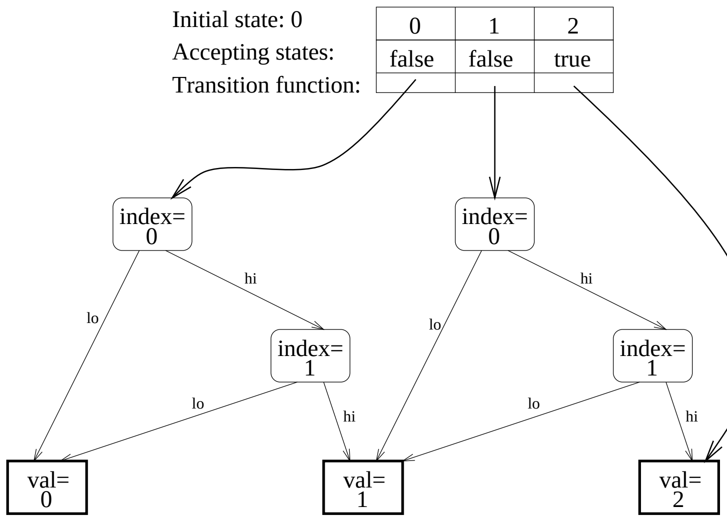
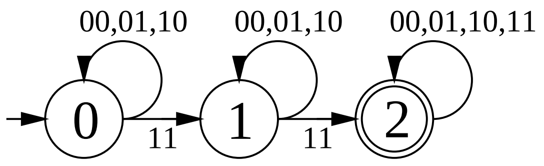


FIGURE 4. BDD automaton representation



could be represented as in Figure 4.

The use of BDDs makes the representation very succinct in comparison to our earlier attempt to handle automata with large alphabets [JJK94]. In most cases, we avoid the exponential blow-up associated with an explicit representation of the alphabet. We shall see that all operations on automata needed can be performed by means of simple BDD operations.

Another possibility would have been to use a two-dimensional array of ordinary BDDs. But that would complicate the operations on automata, because many more BDD operations would be needed.

6.2 Rewriting formulas

The first step in the translation consists of rewriting formulas so as to eliminate nested terms. Then all terms are variables and all formulas are among a small number of basic formulas.

6.3 Translating formulas

The translation is inductive. All automata corresponding to basic formulas have a small number of states (less than five!).

The composite formulas are translated by use of operations on automata. For $\neg\phi$, $\phi_1 \wedge \phi_2$ and $\exists P : \phi$, which are the ones left after rewriting, we need the operations of complement, product, projection, and determinization.

Complement

Complementation is done by simply negating the bit vector representing the set of final states.

Product

The product automaton A of two automata A_1 and A_2 is

$$(Q_1 \times Q_2, \mathbb{B}^k, (i_1, i_2), \delta, F_1 \times F_2),$$

where $\delta((q_1, q_2), b) = (\delta_1(q_1, b), \delta_2(q_2, b))$. We are careful, however, to consider only those states of A that are reachable from (i_1, i_2).

When considering a new state (q_1, q_2), we need to construct the BDD representing the corresponding part of the transition function δ. We use the binary apply operation on the BDDs corresponding to q_1 and q_2. For each pair of states (q', q'') encountered in a pair of leaves, we associate a unique integer in the range $\{0, 1, \ldots N-1\}$, where N is the number of different pairs considered so far. In this way, the new BDDs created conform with the standard representation.

Projection and determinization

Projection is the conversion of an automaton over $\mathbb{B}^{k+1}$ to a nondeterministic automaton over $\mathbb{B}^k$ necessary for translating a formula of the form $\exists P : \phi$. On any letter $b \in \mathbb{B}^k$, there are two transitions possible in the nondeterministic automaton corresponding to whether the P-track is 0 or 1. Therefore this automaton is not hard to construct using the projection (restriction) operation of BDDs.

Determinization is done according to the subset construction. The use of the apply operation is similar to that of the product construction except that leaves hold subsets of states.

6.4 Minimizing

Minimization seems essential in order to obtain an effective decision procedure. For example, if a tautology occurs during calculations, then it is obviously a good idea to represent it using a one-state automaton instead of an automaton with e.g. 10,000 states.

The difficulty in obtaining an efficient minimization algorithm stems from the requirement to keep our shared BDDs in reduced form. Recall that a reduced BDD has no duplicate terminals or nonterminals. Such a BDD is just a specialized form of directed acyclic graph that has been compressed by combining structurally isomorphic nodes (see Aho, Hopcroft, and Ullman [AHU74] or Section 3.4 of Cai and Paige [CP94]). In addition, a reduced BDD has no redundant tests [Bry92]. Such a BDD is obtained by repeatedly pruning every internal vertex v that has both outedges leading to the same vertex w, and redirecting all of v's incoming edges to w.

Suppose that the shared BDD had all duplicate terminals and nonterminals eliminated, but did not have any of its redundant tests eliminated. Then it would be easy to treat the deterministic finite automaton combined with its BDD machinery as a single automaton whose states were the union of the BDD nodes and the original automaton states, and whose alphabet were zero and one. If this derived automaton had n states, then it could be minimized in $O(n \log n)$ steps using Hopcroft's algorithm [Hop71]. Unfortunately, such an automaton would be too big.

For our purposes, the space savings due to redundant test removal is of crucial importance. But the important 'skip' states that arise from redundant test removal complicates minimization. Our algorithm combines techniques based on [AHU74] with new methods adapted for use with the shared BDD representation of the transition function. For a finite automaton with n states and a transition function represented by m BDD nodes, the algorithm presented here achieves worst-case running time $O(\max(n, m)n)$.

Terminology

A *partition* $\mathcal{P}$ of a finite set U is a set of disjoint nonempty subsets of U such that the union of these sets is all of U. The elements of $\mathcal{P}$ are called its *blocks*. A *refinement* $\mathcal{Q}$ of $\mathcal{P}$ is a partition of U such that any block of $\mathcal{Q}$ is a subset of a block of $\mathcal{P}$. If $q \in U$, then $[q]_{\mathcal{P}}$ denotes the block of partition $\mathcal{P}$ containing the element q, and when no confusion arises, we drop the subscript.

Let $A = (Q, \mathbb{B}^k, i_0, \delta, F)$ denote a deterministic finite automaton, and let $\mathcal{P}$ be a partition of Q, and $\mathcal{Q}$ a refinement of $\mathcal{P}$. A block B of $\mathcal{Q}$ *respects* the partition $\mathcal{P}$ if for all $q, q' \in B$ and for all $b \in \mathbb{B}^k$, $[\delta(q, b)]_{\mathcal{P}} = [\delta(q', b)]_{\mathcal{P}}$. Thus, δ cannot distinguish between the elements in B relative to the partition $\mathcal{P}$. A partition $\mathcal{Q}$ *respects* $\mathcal{P}$ if every block of $\mathcal{Q}$ respects $\mathcal{P}$. A partition is *stable* if it respects itself. The *coarsest, stable partition* $\mathcal{Q}$ *respecting* $\mathcal{P}$ is a unique partition such that any other stable partition respecting $\mathcal{P}$ is a refinement of $\mathcal{Q}$.

The refinement algorithm

The minimal automaton A' recognizing $L(A)$ is isomorphic to the automaton defined by the coarsest stable partition $\mathcal{Q}^A$ of Q respecting the partition $\{F, Q \setminus F\}$. The states of A' are $\mathcal{Q}^A$, the transition function δ' is defined by $\delta'([p], b) = [\delta(p, b)]$, the initial state is $[i_0]$, and the set of final states is $F' = \{[f] | f \in F\}$.

Now we are ready to sketch our minimizing algorithm, which works by gradually refining a current partition.

- First split Q into an initial partition $\mathcal{Q} = \{F, Q \setminus F\}$. Note that $\mathcal{Q}^A$ is a refinement of this partition.
- Now let $\mathcal{P}$ be the current partition. We construct the new current partition $\mathcal{Q}$ so that it respects $\mathcal{P}$ while $\mathcal{Q}^A$ remains a refinement of $\mathcal{Q}$.

  For each state q in Q consider the functions $f_q : \mathbb{B}^k \rightarrow \mathcal{P}$ defined by $f_q(b) = [\delta(q, b)]_{\mathcal{P}}$ for all q and b. Now let the equivalence relation $\equiv$ be defined as $q \equiv q' \Leftrightarrow (f_q = f_{q'} \wedge [q]_{\mathcal{P}} = [q']_{\mathcal{P}})$. The new partition $\mathcal{Q}$ then consists of the equivalence classes of $\equiv$. By definition of the f_q's, $\mathcal{Q}$ respects $\mathcal{P}$ and is the coarsest such partition implying the invariant.

  We repeat this process until $\mathcal{P} = \mathcal{Q}$.

It can be shown that the final partition $\mathcal{Q}$ is obtained in at most n iterations and equals $\mathcal{Q}^A$. The preceding algorithm is an abstraction of the initial naive algorithm presented in Section 4.13 of [AHU74].

The difficult step in the above algorithm is the splitting according to the functions f_q. However, we can here elegantly take advantage of the shared BDD representation. The idea is to construct a BDD representing the functions f_q for each state. We represent a partition of the states Q, by associating with each state $q \in Q$ a *block id* identifying its block. The BDD for f_q is calculated by performing a unary apply on the collection of shared BDDs, where the value calculated in a leaf is the block id. By a suitable generalization of the standard algorithm, it is possible to carry out these calculations while visiting each node at most once (assuming that hashing takes constant time). Thus the split operation requires time $O(\max(n, m))$. Since we use shared BDDs, we may use the results of the apply operations directly as new block ids.

The splitting step without hashing

An alternative implementation of the splitting step is possible that achieves the same worst case time bound $O(\max(n,m))$ without hashing. It is instructive to first consider the case in which the shared BDDs are reduced only by eliminating redundant nodes but not by eliminating redundant tests. In this case the BDD may be regarded as an acyclic deterministic automaton D whose states are the BDD nodes, and whose alphabet is zero and one. Consider a partition $\mathcal{P}'$ of the BDD nodes defined by equivalence classes of the following relation. Two BDD leaves are equivalent iff their next states belong to the same block of partition $\mathcal{P}$. All decision nodes of the BDD are equivalent. The coarsest stable partition $\mathcal{Q}'$ that respects $\mathcal{P}'$ for automaton D can be solved in $O(m)$ worst case time by Revuz [Rev92] and Cai and Paige [CP94], Sec. 3.4. Finding the equivalence classes of states in Q that point to BDD roots belonging to the same block of $\mathcal{Q}'$ (i.e., finding the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$) solves the splitting step in the original automaton in time $O(n)$.

In the case of fully reduced BDDs, the splitting step is somewhat harder, and a closer look at the BDD structure is needed. For each decision node v, $v.index$ represents a position in a string of length k such that $v.index < (v.lo).index \wedge v.index < (v.hi).index$. For each BDD leaf v we have $v.index = k$, and let $v.lo = v.hi$ be an automaton state belonging to Q. For each BDD node v we define function $f_v : \mathbb{B}^k \to \mathcal{P}$ much like the way functions f_q were defined earlier on automaton states. For each nonleaf v, f_v is defined by the rule $f_v(b) = f_{v.lo}(b)$ if $b_{v.index} = 0$; $f_v(b) = f_{v.hi}(b)$ if $b_{v.index} = 1$. For each leaf v, f_v is a constant function that maps every argument into an element (i.e., a block) of partition $\mathcal{P}$.

If $q \in Q$ is an automaton state that points to a BDD root v, then, clearly, $f_q = f_v$. It is also not hard to see that for any two nonleaf BDD nodes v and v', $f_v = f_{v'}$ iff either of the following two conditions hold:

1. $v.index = v'.index \wedge f_{v.hi} = f_{v'.hi} \wedge f_{v.lo} = f_{v'.lo}$, or
2. $f_{v.hi} = f_{v.lo} = f_v \wedge v.hi = v'$.

This leads to the more concrete equivalence relation $\equiv$ on BDD nodes defined as $v \equiv v'$ iff $f_v = f_{v'}$ iff either,

1. $v.index = v'.index = k \wedge [v.lo]_{\mathcal{P}} = [v'.lo]_{\mathcal{P}}$, or
2. $v.index = v'.index < k \wedge v.hi \equiv v'.hi \wedge v.lo \equiv v'.lo$, or
3. $v.index < k \wedge v.lo \equiv v.hi \equiv v'$.

Note that two BDD nodes of different index can be equivalent only by condition (3). Note also, that we can strengthen condition (2) with the additional constraint $v.hi \not\equiv v.lo$ without modifying the equivalence relation. These two observations allow us to construct the equivalence classes inductively using a bottom-up algorithm that processes all BDD nodes of the same index in descending order, proceeding from leaves to roots. The steps are sketched just below.

1. In a linear time pass through all of the BDD nodes, place each node in a bucket according to its index. An array of $k+1$ buckets can be used for this purpose.
2. Next, distribute the BDD leaves (contained in the bucket associated with index k) into blocks whose nodes all have *lo* successors that belong to the same block of $\mathcal{P}$. This takes time proportional to the number of leaves.
3. For $j = k-1, ..., 0$ examine each node v with $v.index = j$. Both nodes $v.lo$ and $v.hi$ have already been examined, and have been placed into blocks. Hence, a streamlined form of multiset sequence discrimination [CP94] can be used to place v either in an old block (according to condition (3)) or a new block (according to condition (2)) for nodes whose children belong pair-wise to the same old block.

The preceding algorithm computes the equivalence classes as the final set of blocks in $O(m)$ time. As before, we can use these equivalence classes to find the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$, which solves the splitting step in the original automaton, in time $O(n)$. Thus, the total worst-case time to solve the splitting step is $O(\max(n,m))$ (without hashing).

In an efficient implementation of finite-state automaton minimization, when the splitting algorithm above is is performed repeatedly, we only need to perform the first step of that algorithm (i.e., sorting BDD nodes according to index) once. Thus, the full DFA minimization algorithm runs in worst case time $O(\max(n,m)n)$ without hashing.

BDD reduction without hashing

Sieling and Wegener[SW93] were the first to compress an arbitrary BDD into fully reduced form in linear time. Their result depended on a radix sort, which is closely related to the multiset discrimination technique that we use. However, their algorithm needs to maintain integer representations of BDD nodes, and it utilizes two arrays of size m. We can show how our algorithm just described can be modified to fully reduce an arbitrary BDD in worst case time linear in the number of BDD nodes (without hashing), but with expected auxiliary space k times smaller than Sieling and Wegener's algorithm.

Let $\mathcal{Q}'$ be the partition of BDD nodes produced by the algorithm. The states of the reduced BDD are the blocks in $\mathcal{Q}'$. For each block $B \in \mathcal{Q}'$,

$B.index$ is the largest index of any BDD node contained in B. Let v' be any node belonging to B of maximum index. If v' is a BDD leaf, then B is a leaf in the reduced BDD (i.e., $B.index = k$), and $B.lo = B.hi = v'.lo$. Otherwise, $B.lo = [v'.lo]_{Q'}$ and $B.hi = [v'.hi]_{Q'}$. The hi and lo successor blocks can be determined during the multiset sequence discrimination pass when a new block is first created. The index of the first node placed in a newly created block is the index for that block.

What distinguishes our algorithm from that of Sieling and Wegener is that our buckets in steps (2) and (3) are associated with actual BDD nodes (inside the main BDD data structure). Their buckets are associated with components of two auxiliary arrays of size m each. If we replaced each equivalence class by a single witness (as they do) each iteration of step (3), then our auxiliary space would be bounded by the maximum number of BDD nodes that have the same index. If BDD nodes were uniformly distributed among indexes, then this number is m/k, which would give us a k-fold advantage in auxiliary space over their algorithm. We expect a minor constant factor advantage in time as well, because our BDD nodes are represented by their locations instead of by computed integer values, and because we avoid array access in favor of less expensive list and pointer processing.

Work is in progress for exploring the "processing the smaller half" idea found in e.g. [PT87]. We should mention, however, that the current implementation of the minimization algorithm in practice seems to run faster than the procedures for constructing product and subset automata.

6.5 MONA *features*

MONA is enriched by facilities similar to those of programming languages.

Predicates

The user may declare predicates that can later be instantiated. For example, if the predicate P is declared by $P(X, x) = (0 = x \wedge x \in X)$, then P can be instantiated as the formula $P(\complement Y, p \oplus 1)$ with the obvious meaning.

Libraries

MONA supports creation of user-defined libraries of predicates.

Separate translation

MONA automatically stores the automaton for a translated predicate. If there are n free variables, then there may be up to $n!$ different automata corresponding to different orderings of variables in the BDD representation.

6.6 To be done

In the current implementation, variables are ordered in their BDDs according to the level of syntactic nesting in the formula; i.e. innermost variables receive the highest index. This strategy is obviously often far from optimal and we are working on implementing heuristics to improve variable ordering. Another orthogonal optimization strategy is to reorder the product constructions by heuristics. In both cases, however, it is not hard to see that finding optimal orderings is NP-complete.

Acknowledgements

We are thankful to Vladimiro Sassone for comments on an earlier version, and to Andreas Potthoff for his advice based on the M2L implementation at the University of Kiel.

7 References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[BK95] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. Technical Report RS-96-7, BRICS, 1995. To appear in CAV '95 Proceedings.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

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

[BSV93] F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 29–40, 1993.

[CP94] J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *to appear Theoretical Computer Science*, 1994. also, U. of Copenhagen Tech. Report, DIKU-TR Num. D-209, 94/16, URL ftp://ftp.diku.dk/diku/semantics/papers/D-209.ps.Z.

[CR94] M-M Corsini and A. Rauzy. Symbolic model checking and constraint logic programming: a cross-fertilisation. In *5th. Europ. Symp. on Programming, LNCS 788*, pages 180–194, 1994.

[GF93] A. Gupta and A.L. Fisher. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 15–28, 1993.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and Paz A., editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.

[JJK94] J. Jensen, M. Jørgensen, and N. Klarlund. Monadic second-order logic for parameterized verification. Technical report, BRICS Report Series RS-94-10, Department of Computer Science, University of Aarhus, 1994.

[KM89] B. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proc. Eigth Symp. Princ. of Distributed Computing*, pages 239–247, 1989.

[PT87] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.

[Rev92] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[RS93] J-K. Rho and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 123–137, 1993.

[Ste93] M. Steinmann. Übersetzung von logischen Ausdrücken in Baumautomaten: Entwicklung eines Verfahrens und seine Implementierung. Unpublished, 1993.

[SW93] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *IPL*, 48:139–144, 1993.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.