Monadic Second-order Logic for Parameterized Verification

Jakob Jensen
Michael Jørgensen
Nils Klarlund

BRICS Report Series RS-94-10

ISSN 0909-0878 May 1994

MONADIC SECOND-ORDER LOGIC FOR PARAMETERIZED VERIFICATION

JAKOB JENSEN, MICHAEL JØRGENSEN, AND NILS KLARLUND
DEPARTMENT OF COMPUTER SCIENCE
BRICS*, UNIVERSITY OF AARHUS
NY MUNKEGADE
DK-8000 AARHUS C.

Abstract. Much work in automatic verification considers families of similar finite-state systems. But an often overlooked property is that sometimes a single finite-state system can be used to describe a parameterized, infinite family of systems. Thus verification of unbounded state spaces can take place by reduction to finite ones.

The purpose of this article is to introduce Monadic Second-order Logic as a practical means of carrying out such reductions. The logic is a highly succinct alternative to the use of regular expressions. We have built a tool that acts as a decision procedure and translator to DFAs.

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.

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

The corresponding author is Nils Klarlund. E-mail: klarlund@daimi.aau.dk.
* Basic Research in Computer Science, Centre of the Danish National Research Foundation.

positions and subset of positions in the input 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 [7]), 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 input.

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.

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

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

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 $L_{a\mathrm{after}b}$ 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 $L_{a\mathrm{after}b}$ 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 6) 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.

Our results. In this article, we present a translator from M2L to DFAs. We discuss potential applications to text processesing and to 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 establishing safety and liveness properties for a non-trivial version of the Dining Philosophers' problem as proposed in [4] by Kurshan and MacMillan.

Comparisons to other work. Parameterized circuits are described using BDDs in [3]. This method relies on formulating inductive steps as finite-state devices and does not provide a single specification language. The work in [5] is closer in spirit to our method in that languages of finite strings are used although not as part of a logical framework. In [1], another approach is given based on iterating abstractions. The parameterized Dining Philosopher's problem is solved in [4] by a finite-state induction principle.

A tool for M2L on finite, binary trees has been developed at the University of Kiel [6]. Apparently, this tool has not been used for verification purposes.

In [2], a programming language for finite domains based on a fixed point logic is described and used for verification of non-parameterized finite systems.

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. Finally, we discuss future work in Section 7.

2. The Monadic Second-order Logic on Strings.

The syntax and semantics of the logic are defined as follows. Let Σ be the input alphabet. We assume that the input string $w \in \Sigma^*$ has length n and is $w = a_0 a_1 ... a_{n-1}$. The *positions* in w are then $0,...,n-1$.

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 $j + i \mod n$, where j is the interpretation of t); or

- of the form $t \ominus i$ (which denotes the position $j - i \mod n$, where j is the interpretation of t);

(Position terms are only interpreted for non-empty input 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 is interpreted as $a_i = a$, where i is the interpretation of t);
- $'a'(T)$ (which is interpreted as $a_i = a$, 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);

A closed formula ϕ denotes a language $L(\phi)$ of the input strings that make ϕ true.

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 [7] for more details). Note that any formula ϕ can be interpreted given an input 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 now that a value assigment and the input string may be described as a word in an alphabet extended with extra tracks that describe the value assignment. 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 an input 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 input string $w = abaa$ and value assigment

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

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

$$\underset{0\,1\,2\,3}{1010}$$

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*, which may be depicted as:

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

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

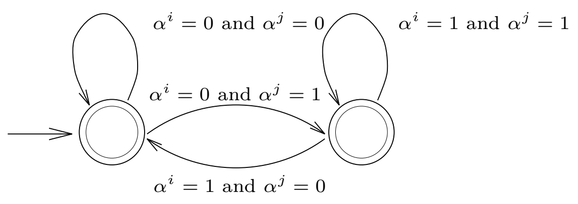
This correspondence can be generalized to any w and any value assigment for a set of variables $\mathcal{P}$ (which can all be assumed to be second-order).

By structural induction on formulas, we construct automata $A^{\phi,\mathcal{P}}$ on 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 $A^{\phi,\mathcal{P}}$:


$\alpha^i = 0$ and $\alpha^j = 0$
$\alpha^i = 1$ and $\alpha^j = 1$
$\alpha^i = 0$ and $\alpha^j = 1$
$\alpha^i = 1$ and $\alpha^j = 0$

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 .6 seconds. (The machine used is an HP700 work station. The intermediate automaton with the largest number of states has 24 states and the size of its transition relation, that is, the number of memory cells needed for its representation, is 228.)

The language $L_{a\mathrm{after}b}$ 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 $L_{<1\mathrm{apart}}$ 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 $L_{<1\mathrm{apart}}$ 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, and the set P_{-1} corresponding to prefixes where the number of a's is one less than the number of b's:

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

The resulting four-state automaton is calculated in 62 seconds. The largest intermediate automaton has 128 states and a transition relation of size 17k. This example exhibits the worst computation time of the small, natural text pattern problems that we have looked at.

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 and thereby precisely to describe the language by interpreting 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

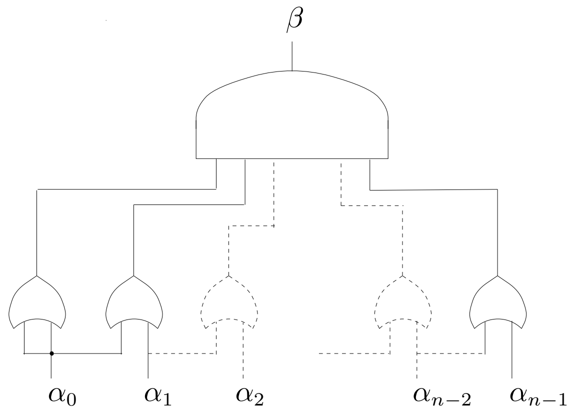


Figure 1. A parameterized circuit.

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

The resulting automaton has three states and is produced in 1.4 seconds. It accepts the language $(1 \cup 10)^*$, which is the regular expression that one would obtain by reasoning about the circuit.

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 on our M2L tool does indeed produce an automaton accepting Σ^* in 0.8 seconds, and thus verifies our claim. (The largest intermediate automaton has 22 states and size 108.)

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 [4] used such a method to verify safety and liveness properties of a non-trivial version of the Dining Philosophers example.

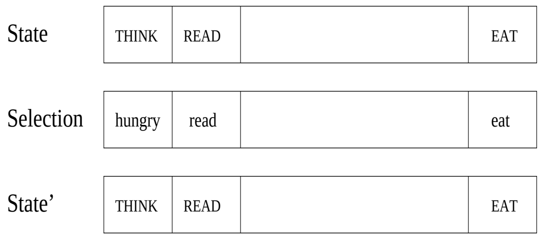


Figure 2. Dining Philosophers with Encyclopedia

In this system, symmetry is broken by a encyclopedia that circulates among the philosophers. Thus each philosopher is in one of three states: EAT, THINK, or READ. The global state can be described as a string *State* of length n over the alphabet $\Sigma_{\mathrm{State}} = \{\mathrm{EAT}, \mathrm{THINK}, \mathrm{READ}\}$, 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 $\Sigma_{\mathrm{Selection}} = \{\mathrm{eat}, \mathrm{think}, \mathrm{read}, \mathrm{hungry}\}$. Thus the selection can be viewed as a string *Selection* over $\Sigma_{\mathrm{Selection}}$, 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 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 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

$$\begin{aligned}
&\mathrm{EAT}_p(\mathit{State}) \equiv p \in P \ \wedge \ p \in Q\\
&\mathrm{READ}_p(\mathit{State}) \equiv p \notin P \ \wedge \ p \in Q\\
&\mathrm{THINK}_p(\mathit{State}) \equiv p \notin P \ \wedge \ p \notin Q
\end{aligned}$$

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 (in practice, we use the UNIX macro tool m4 to write down the formulas).

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 [4] both properties are proved in terms of a complicated induction hypothesis. This hypothesis is itself a distributed system, where each process has four states. (The Liveness Property in [4] 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:

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

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

$$\forall \mathit{State}, \mathit{State}', \mathit{Selection} : \phi(\mathit{State}) \ \wedge \ \mathrm{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \Rightarrow \phi(\mathit{State}'),$$

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.

$$\exists! p : \mathrm{READ}_p(\mathit{State}),$$

are invariant. The verification of each formula takes two and a half minutes. (The largest intermediate automaton has about 200 states and size about 18k.)

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.

Establishing the Liveness Property. The Liveness Property can be expressed in Temporal Logic as

$$(1) \qquad \Box\,(\mathrm{READ}_{p\ominus 1} \ \Rightarrow \ \Diamond\mathrm{READ}_p),$$

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:

$$(2) \qquad \Box\,(\mathrm{EAT}_p \ \Rightarrow \ \Diamond\neg\mathrm{EAT}_p).$$

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

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

trans($State, State', Selection$) $\equiv$
$\forall p$:

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

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

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

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

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

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

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

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

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

Figure 3. The transition relation

- Case EAT_p holds. By asssumption (2), there is an instant when $\text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_p$ holds. Thus if

  (3) $$\text{READ}_{p\ominus 1} \;\wedge\; \text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_p \Rightarrow \circ\text{READ}_p$$

  is a valid property of the transition system, $\Diamond\text{EAT}_p$ holds. In fact, we verified using our tool that (3) indeed holds.
- Case $\neg\text{EAT}_p$ holds. If EAT_p becomes true, then use the previous case. Otherwise, $\neg\text{EAT}_p$ continues to hold. Now, by the assumption (2) at some point $\neg\text{EAT}_{p\oplus 1}$ will hold. We then use the property

  (4) $$\text{READ}_{p\ominus 1} \;\wedge\; \neg\text{EAT}_p \;\wedge\; \neg\circ\text{EAT}_{p\oplus 1} \Rightarrow \circ\text{READ}_p \;\vee\; \circ\text{EAT}_p,$$

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

6. Implementation.

Our implementation is written in C. We chose explicit garbage collection, which substantially complicated the programming.

We discuss next how formulas without the input predicates $'a'(p)$ and $'a'(P)$ are translated to automata. Thus the alphabets considered are of the form $\mathbb{B}^k$, where k is the number of free variables. Each $b \in \mathbb{B}^k$ is called an *extension*.

The most obvious choice for representing the transition relation would be by list structures that for each pair of states (s, s') detail the set of extended letters b such that (s, b, s') is a transition. Unfortunately, this representation has an exponential blow-up in the number of free variables of ϕ.

Our solution to this problem is to give a compact representation of a set of extensions $E \subseteq \mathbb{B}^k$, without necessarily mentioning every extension explicitly. This can be done the following way : two extensions 01 and 00 can be expressed as an *extension expression* 0x, where x is read as 0 or 1. On the other hand, 01 and 10 cannot be compressed this way. Using this technique, we can express an extension set E as a list of extension expressions

$$E = (e_1, \ldots, e_n), \text{ where } e_i \text{ is of the form } u_1 \cdots u_k, u_j \in \{0, 1, x\}$$

If moreover $e_i \cap e_j = \emptyset$, $i \neq j$, the set of expressions is said to be in *exclusive normal form.* We use this form to simplify the computations involved in Boolean operations on extension expression lists. Specifically, we use the identity $E \backslash e = E \cap \overline{e} = (e_1 \backslash e, \cdots, e_n \backslash e)$.

We represent a transition relation as a set of transitions of the form (s_1, E, s_2), where E is an extension expression list in exclusive normal form. In Figure 4, the automaton for the formula $p < q$ and the transition relation of the corresponding automaton are shown.

With this representation, all transitions from state s to state s' are readily found once s and s' have been located. Our algorithms work by processing pairs (s, s') in the order they appear in the list structures.

Note that our extension representation has a potentially exponential blow-up, that is, there are extension sets on k variables that require approximately 2^k extension expressions for their representation. Fortunately, our experiments have given evidence that this often does not happen in practice.

Automata operations. To keep Boolean operations on automata simple, we have chosen to use deterministic automata. All Boolean operations can be implemented using only two basic operations : complement and cross product. In addition, we need a projection operation to handle existential quantification and the subset construction.

Complementation. Since all automata are deterministic, the complement automaton $A = \neg A_1$ is found simply by switching final and non-final states.

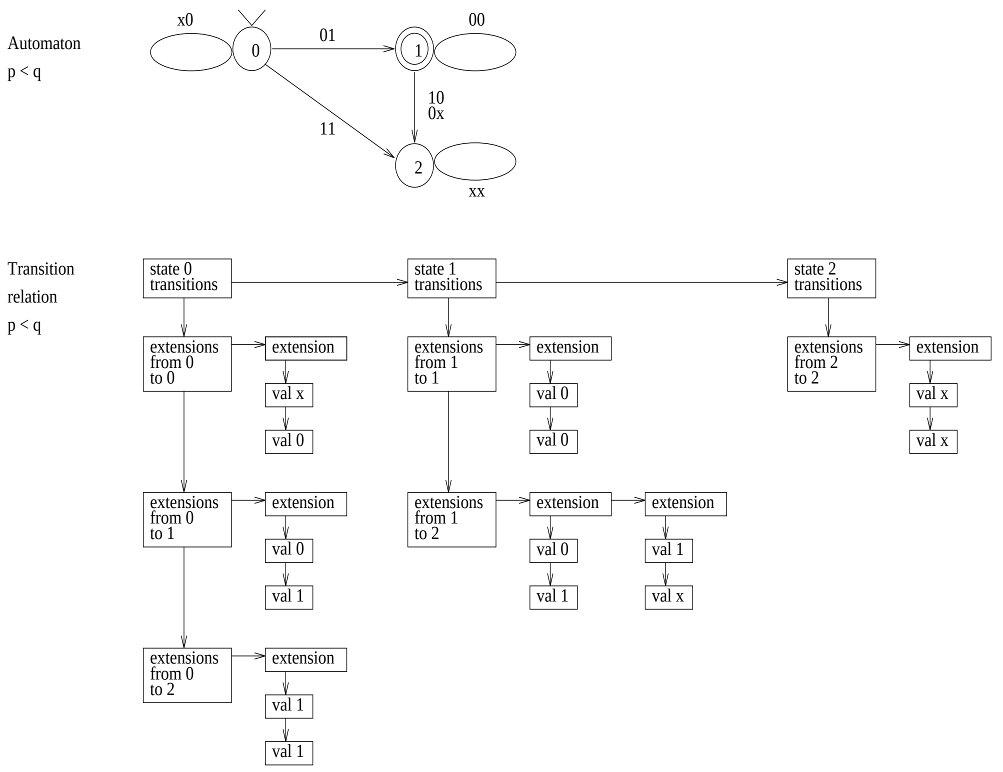


Figure 4. Representation of transition relation.

Cross product. The transition relation of the cross-product automaton $A = A_1 \times A_2$ is calculated as follows. Given two states $s = (s_1, s_2)$ and $s' = (s'_1, s'_2)$ we find the set of extensions E leading from s to s' in A as $E = E_1 \cap E_2$, where $(s_1, E_1, s'_1) \in \rightarrow_1$ and $(s_2, E_2, s'_2) \in \rightarrow_2$. If $E = \emptyset$, of course we do not represent the transition. Since we only want to find states reachable from the the start state $s^0 = (s^0_1, s^0_2)$, we calculate all transitions recursively from the start state. The set of final states are easily found, using the definition $S^F = S^F_1 \times S^F_2$.

Determinization and projection. We use the subset construction to determinize. Only reachable subset states are constructed. In practice, the blow-ups appear to be mostly benign. Determinization is needed only in connection with quantifiers, which effect the removal of the track corresponding to the free variable.

In the example in Figure 5, the result of removing the second track is a non-deterministic automaton, because there are two different transitions on input 1. The transitions originating in the subset state $\{s_2, s_3\}$ are calculated as follows. If (s_2, E_2, s_4) and (s_3, E_3, s_5) are transitions belonging to the determinized transition relation, then $(\{s_2, s_3\}, E, \{s_4, s_5\})$ is a transition, if $E = E_2 \cap E_3 \neq \emptyset$. Thus, by Boolean operations on extension expression sets, we are able to compute the transition relation of the subset automaton.

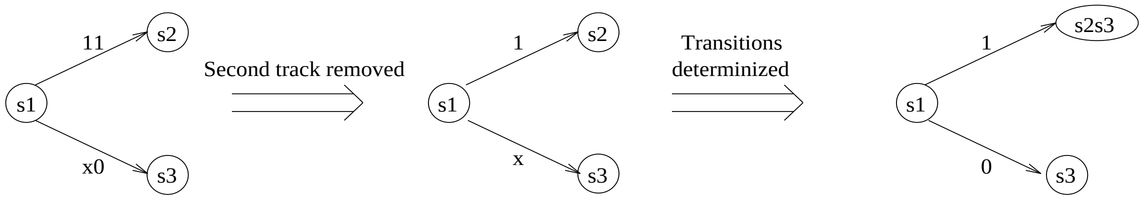


Figure 5. Determinisation.

Minimization. Unfortunately the result of Boolean operations on automata is not always a minimal automaton.

Minimizing an automaton is done by "collapsing" classes of equivalent states into single states. Intuitively, two states s_1 and s_2 are equivalent, and we write $s_1 \equiv s_2$, if the set of strings accepted from s_1 is exactly the set of strings accepted from s_2. To find classes of equivalent states, we initialize an equivalence relation consisting of two classes, final states and non-final states, and we subsequently refine the relation until no more classes are found. This is done according to the requirement: $s_1 \equiv s_2$ implies for all $b \in \mathbb{B}^k$, $\delta(s_1, b) \equiv \delta(s_2, b)$, where δ is the transition relation represented as a function. In our representation, this condition becomes

$$\begin{array}{l} s_1 \equiv s_2 \Rightarrow \\ \quad \text{for all transitions } (s_1, E_1, s_1') \text{ and } (s_2, E_2, s_2') : \\ \quad\quad E_1 \cap E_2 \neq \emptyset \Rightarrow s_1' \equiv s_2' \end{array}$$

Our algorithm is an adaptation of the straightforward n^2 text book method. When a pair of states s_1 and s_2 have been declared inequivalent, our algorithm checks all other pairs s_1' and s_2' that are still considered equivalent to see whether they now are inequivalent with respect to s_1 and s_2, that is if $E \cap F \neq \emptyset$, where (s_1', E, s_1) and (s_2', F, s_2) are transitions. Thus the total running time becomes $m^2 \cdot n^4$, where m bounds the size of the extension expression list and n is the size of the state space. Note, however, that when m stays close to n our algorithm is exponentially faster than any convential algorithm that is based on an explicit representation of the alphabet. Also, for sparse transition systems (which are the most common), the running time is only $m^2 \cdot n^2$.

Handling of first-order variables. As in [6], we treat first-order variables as if they were second-order variables, except when they are eliminated together with their quantifier. Then we impose the condition that the second-order variables contain exactly one element.

Compression of extensions. In general, it is an NP-complete problem to minimize the representation of an extension expression list. We have implemented an algorithm, that reduces extension expression lists so that no two extension expressions e_i and e_j exist that can be compressed into one. This does not produce a minimal extension expression list, but reduces extension expression lists by 5 to 40% in practice.

7. Discussion

We have shown that for a non-trivial distributed system, our invariant method for boiling down an unbounded state space to a finite one is a promising alternative to the use of induction. These results were obtained on a preliminary implementation of a M2L to DFA translator that can be substantially improved in many ways:

- The size of the transition relation could be reduced by an order of magnitude if extension expressions are packed into machine words. Running time would also be an order of magnitude faster.
- BDDs can be used to represent the transition function so as to obtain an $m^2 \cdot n^2$ minimization routine.
- In order to avoid unnessary combinatorial explosions, heuristics for transforming the formula should be introduced.
- A library of common predicates and their corresponding DFAs would improve efficiency.
- There are evident ways of parallelizing our tool by sending separate subformulas to different machines.

Work is in progress to implement some of these ideas.

Acknowledgements. We are thankful to Andreas Potthoff for his advice based on the M2L implementation at the University of Kiel.

References

1. 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.
2. 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.
3. 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.
4. B. Kurshan and K. MacMillan. A structural induction theorem for processes. In *Proc. Eigth Symp. Princ. of Distributed Computing*, pages 239–247, 1989.
5. 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.
6. M. Steinmann. Übersetzung von logischen Ausdrücken in Baumautomaten: Entwicklung eines Verfahrens und seine Implementierung. Unpublished, 1993.
7. 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.