Good paper about the context of the complementation of non-deterministic omega-automata. It is used for testing language containment, which is used in the language containment approach of automata-theoretic model checking (property specified as automaton; is the language represented by the system automaton contained in the language represented by the property automaton?
—> See Vardi, 2007, "Automata-Theoretic Model Checking Revisited"

Language containment of non-deterministic ω-automata

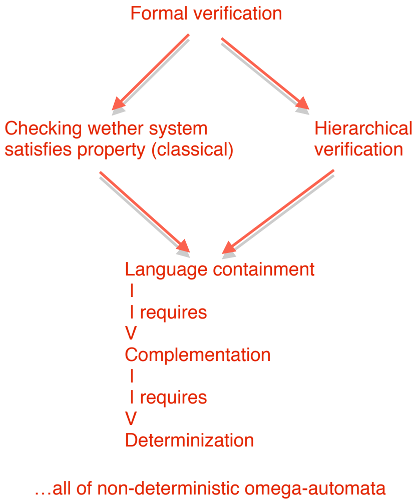
Serdar Taşıran[*] Ramin Hojati Robert K. Brayton
Email: {serdar,hojati,brayton}@ic.eecs.berkeley.edu

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley

Abstract. Algorithms and techniques to determinize and complement ω-automata with various forms of fairness constraints are investigated and implemented. A tool-box is constructed by supplementing these algorithms with less complex ones for certain special cases. Recently published constructions which are asymptotically optimum constitute some of the core routines. The principal use of these tools is in checking language containment between two non-deterministic automata. In language containment based verification, the need for this check may arise in two occasions: when checking whether a system satisfies a property expressed as a non-deterministic automaton, or, in hierarchical verification, where the more detailed system description must satisfy the more abstract specification.

acceptance conditions

We give examples motivating the utility of non-deterministic specifications and complexity results relating non-deterministic and deterministic ω-automata. The algorithms mentioned have been implemented as part of the HSIS verification system. Experimental results are presented to demonstrate the practical applicability of the algorithms.


Formal verification

Checking wether system satisfies property (classical)

Hierarchical verification

Language containment
I
I requires
V
Complementation
I
I requires
V
Determinization

…all of non-deterministic omega-automata

1 Introduction

Many problems in formal verification can be formulated in a language containment framework. Deciding whether the language of an automaton M is contained in that of P, i.e., $L(M) \subseteq L(P)$, is PSPACE-complete if P is non-deterministic. Language containment is usually tested by converting to a language emptiness question: an automaton $\overline{P}$ accepting the language $\overline{L(P)}$ is constructed, and it is determined whether $L(M \times \overline{P})$ is empty. Known asymptotically optimum ways of obtaining such a $\overline{P}$ require determinizing P first.

Determinization of ω-automata with fairness constraints is more complex than determinization of finite automata. The straightforward subset construction does not work; specialized constructions taking into account the particular kind of fairness constraints need to be employed. Because of the exponential cost of determinization, it has been considered impractical, and verification systems have either been restricted to using deterministic automata or employed means of verification other than language containment([Kur87],[HSIS94])

[*] Supported by SRC under grant DC-008-026.

Nevertheless, non-determinism is a valuable tool in automata theory, as well as hardware verification. It offers the versatility of having concise specifications, and captures the structure of certain languages or sequences of events in a natural way. The flexibility to use non-deterministic automata as specifications makes certain operations such as variable hiding or abstraction much easier. Therefore, it is useful to have a tool that handles non-deterministic automata and hides from the user the complexity of conversion to deterministic automata. This provides the motivation for our study.

The main product of our work is a tool for determinizing and complementing various kinds of ω-automata. The contributions reported in this paper are

- A partly implicit implementation of Safra's construction ([Saf88]) for determinizing Büchi automata and determinization routines for Streett and Rabin automata by converting them to Büchi automata first.
- Specialized less complex constructions for the following kinds of automata and their implicit implementations.
  - Automata with fairness constraints of the form $\bigvee_{1 \leq i \leq h} G^{\infty} A_i$ (i.e., a run must eventually fall in one of the A_is to be accepted).
  - k-step observably non-deterministic automata (See [Cer92] and Section 4.4.)
- Experimental results demonstrating the practicality of determinization as part of a verification tool.

Section 2 defines the concepts and notation. In Section 3 we provide more concrete motivation for our work through examples and describe the applications of language containment of non-deterministic ω-automata. Section 4 details the main body of our work on determinization and complementation. Previous work and complexity results are given, and the algorithms mentioned above and techniques used in their implementation are explicated. In Section 5, the performance of the algorithms on various examples is studied and the limitations of the implementation are addressed. In Section 6 we summarize our work and propose future directions.

2 Preliminaries: ω-automata

An *ω-automaton* over an alphabet Σ consists of a finite transition structure and a fairness condition. More precisely, it is a 5-tuple (Q, Σ, T, I, Φ), where Q is the set of states, Σ is the alphabet, $T \subseteq Q \times \Sigma \times Q$ is the transition relation, $I \subseteq Q$ acceptance condition is the set of initial states, and Φ is the fairness condition. A *run* on an input string $x = x_0 x_1 ... \in \Sigma^{\omega}$ is an infinite sequence of states $\sigma = s_0 s_1 ...$, starting in one of the initial states ($s_0 \in I$) and making transitions that are allowed by T ($\forall i \; (s_i, x_i, s_{i+1}) \in T$). An automaton is said to be *deterministic* if it has only one initial state, and on any input symbol there is exactly one transition possible from any given state. A deterministic automaton has exactly one run on any given input string. The fairness condition is a condition on the *infinitary set* of a run $inf(\sigma)$, i.e., the set of states that are visited infinitely often by σ. The *infinitary part* of a run is the latter part of it during which no state outside

$inf(\sigma)$ is visited. A run σ is said to be accepting if $inf(\sigma)$ satisfies Φ. A string is accepted by an ω-automaton if and only if it has an accepting run.

The fairness constraint is expressed conveniently as a Boolean combination of the following: $G^{\infty}A$, meaning that the run must eventually fall completely inside A, i.e. $inf(\sigma) \subseteq A$, and $F^{\infty}A$, meaning that the run must visit some state in A infinitely often, i.e. $inf(\sigma) \cap A \neq \emptyset$. Some commonly used forms of fairness constraints are

- $\Phi = F^{\infty}A$ (Büchi Fairness)
- $\Phi = \bigwedge_i (F^{\infty}L_i \Rightarrow F^{\infty}U_i) = \bigwedge_i (G^{\infty}\overline{L_i} \vee F^{\infty}U_i)$ (Streett Fairness)
- $\Phi = \bigvee_i (F^{\infty}L_i \wedge G^{\infty}U_i)$ (Rabin Fairness)
- L-processes and L-automata have fairness constraints on edges as well as states ([Kur87]). These automata can be translated to the above efficiently.

$L(A)$, the *language accepted by* A, is defined as the set of ω-strings that are accepted by the automaton A. A language L is called *ω-regular* if it can be written as $L = \bigcup_{i=1}^{n} U_i V_i^{\omega}$ for regular languages U_i and V_i. Non-deterministic Büchi, Streett and Rabin, and deterministic Streett and Rabin automata all accept exactly the class of ω-regular languages, whereas deterministic Büchi automata are strictly less expressive.

3 Motivation

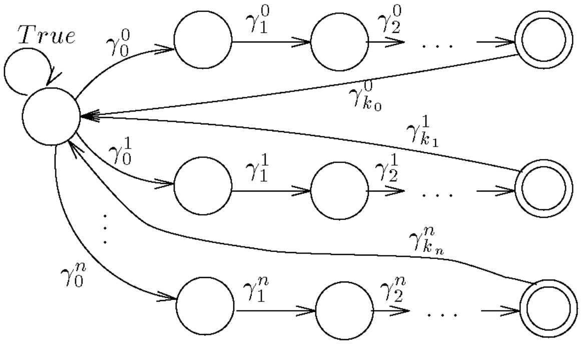
Advantages of specifying properties non-deterministically

An automaton is said to be *non-deterministic* if it has more than one initial state, or from a certain state, on an input symbol, there is more than one possible transition. This freedom can be used to advantage in various ways. It is known that non-deterministic automata are at least exponentially more concise. Moreover, in many cases, non-deterministic automata provide a simpler and more intuitive means for specifying a language. One common case is looking for certain substrings in a string, i.e., pattern matching. Suppose we want to construct a deterministic automaton accepting a language of the form

$$L(S) = \{x \in \Sigma^{\omega} \mid S \subset \Sigma^{*}, \exists \gamma \in S \text{ such that } \gamma \text{ occurs in } x \text{ infinitely often}\}$$

i.e., we want to match one of a finite number of patterns. Figure 1 shows the form of a non-deterministic automaton that expresses this property naturally. In general, a deterministic automaton recognizing a language of this form must take into account the common substrings of the strings in S, and must store information about a sufficient number of previous symbols to be able to decide whether the current input symbol results in a match.

Now consider a property verification example where we would like to prove the following about a communication channel: "It is infinitely often the case that the data transmitted is eventually received correctly at the other end." The automaton in Figure 2 describes this property. A deterministic automaton expressing the same property must make use of special constructs to keep track of transmissions that are never received and those that are eventually received. (See [Taş95] for the details and other examples.)



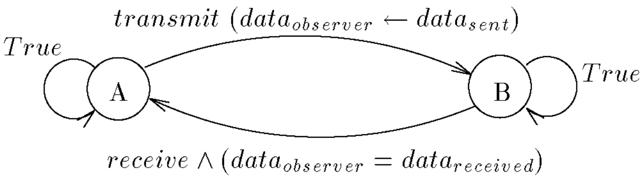
Accepts a word if it contains infinitely many occurrences of y0, y1,...,yn

Fig. 1. Non-deterministic Büchi automaton accepting $L(S)$, where $S = \{\gamma^0, \gamma^1,$-$..., \gamma^n\}$, and γ^i_j denotes the jth symbol in string γ^i. The final states are marked with double circles.

To benefit from the flexibility offered by non-determinism, formal verification systems must incorporate algorithms that can handle them. In the rest of this section, we investigate the problem of language containment using non-deterministic automata.

A formal verification environment based on language containment models a system M as a collection of interacting ω-automata: $M_1, ..., M_n$ [Kur92]. The need for language containment typically arises in one of the following schemes:

1. A property, specified as an ω-automaton P, is to be proved about the system. This is done by checking whether or not the language of the system $L(M) \overset{def}{=} L(M_1) \times L(M_2) \times ... \times L(M_n)$ is contained in the language of the property automaton, $L(P)$.
2. There are several representations of the system, or several phases of the design process, which differ in the level of abstraction used to specify the system (*hierarchical specification*). A desirable feature of a hierarchical specification is that, if properties are proved about an abstract version of the system ($L(M_{abs}) \subseteq L(P)$), they are satisfied by the more detailed, refined version ($L(M_{rfn}) \subseteq L(P)$). A sufficient condition for this is $L(M_{rfn}) \subseteq L(M_{abs})$. Much of the time, refinement is done module by module, i.e., if $M_{abs} = \prod_i M_{abs,i}$, later on $M_{abs,i}$ is replaced by $M_{rfn,i}$. The check to be performed in this case to verify whether property $L(P)$ carries over to the refined system if, for all i, $L(M_{rfn,i}) \subseteq L(M_{abs,i})$. A more abstract representation of a module may be employed either to reduce the complexity of



$transmit\ (data_{observer} \leftarrow data_{sent})$
$True$
A
B
$True$
$receive \wedge (data_{observer} = data_{received})$

Fig. 2. Non-deterministic Büchi automaton describing the property "It is infinitely often the case that the data transmitted is eventually received correctly at the other end." The fairness constraint is $F^\infty\{\mathrm{A}\} \wedge F^\infty\{\mathrm{B}\}$.

verification by hiding unnecessary detail or because some components of the system have not been designed yet, therefore, little is known about their behavior. This would often be the case in modelling the environment of a hardware module.

In the first context, non-determinism can arise because property automata may be non-deterministic. At some level of abstraction, it may be easier to express a property using a non-deterministic automaton. A property of the form “if event e happens, then one of the acceptable responses in S must eventually follow” can be expressed naturally using non-determinism and may be hard to express otherwise.

Second application of language containment

Now consider the second case. The following is an example of abstraction and subsequent refinement, where we would like to verify a system consisting of a resource, a number of users wanting to access it, and an arbiter that allocates the resource to them, one at a time. When verifying such a system, we may not know the details of the arbiter, but we may know that it will eventually be designed to service each user in a fair fashion, i.e., each user will gain use of the resource infinitely often. We can model this arbiter as an automaton that, at each step, non-deterministically allocates the resource to some user that requests it, and that also has the fairness condition that each user be serviced infinitely often. We may be able to verify some properties about the system with this much information. At a later phase of the design, we may have a more refined model for the arbiter. If we can prove that the languages of the refined modules are contained in those of the more abstract ones, the properties proved at the more abstract level will still hold. The following demonstrates this scheme in more detail for an example system.

Example 1. Let the abstract model of the arbiter mentioned be M_{arb} in Figure 3. A and B are the function units that want to access the resource. The structure of A is shown in Figure 3, B's is the same.

It is possible to prove the following properties of the system consisting of the abstract models of the function units and the controller.

1. At most one functional unit can be using the resource at any given time.
2. If it is infinitely often the case that a functional unit asserts *request* for two clock periods in a row, then it gets the resource infinitely often.

Consider now the actual implementation of the controller shown in Figure 3. It can be shown that the language of the implementation is contained in that of the abstract model. Assuming that the function units are implemented in a way that the same language inclusion holds, the properties above hold for the actual implementation of the system also.

In both applications of language containment mentioned, the check $L(M) \subseteq L(P)$ requires an automaton $\overline{P}$. The complementation operation for obtaining $\overline{P}$ from P is trivial if P is deterministic: only the fairness constraint is complemented. However, if P is non-deterministic, this approach will not work, as a given input string may have both accepting and non-accepting runs. All known

Complementation of a deterministic omega-automata: just complement acceptance condition

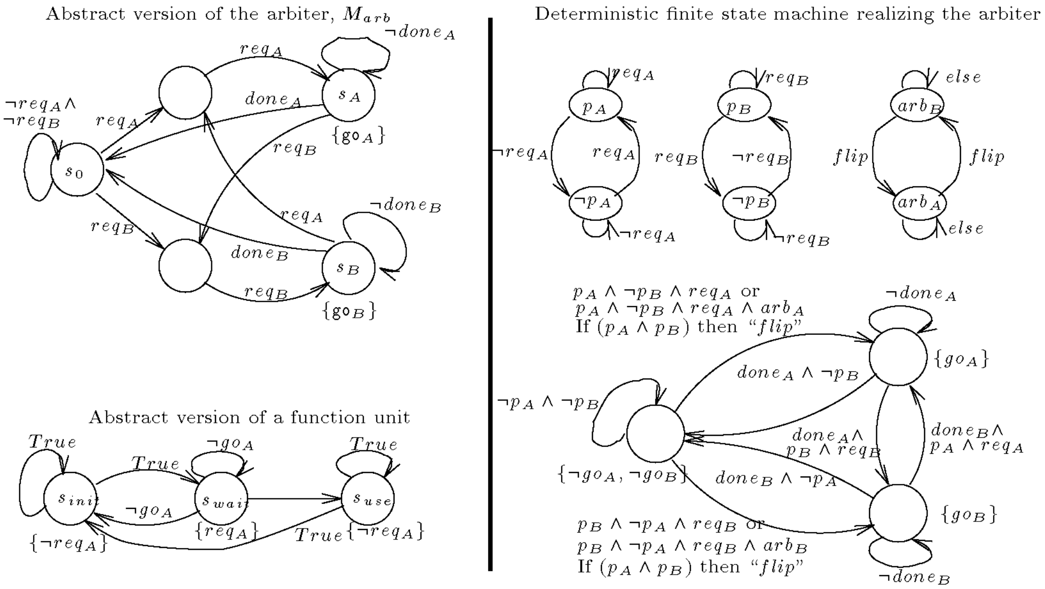


Fig. 3. A hierarchical verification example. The possible outputs at each state are indicated in braces next to the state. For the abstract version of the arbiter, the fairness constraint is that both A and B are serviced infinitely often and that neither of them holds the resource indefinitely, i.e. $\Phi = F^{\infty}s_A \wedge F^{\infty}s_B \wedge F^{\infty}\overline{s_A} \wedge F^{\infty}\overline{s_B}$. For the other machines, there are no fairness constraints, i.e., all runs are accepting.

asymptotically optimum ways to perform the complementation operation involve determinizing the automaton first. The next section elaborates on this point and presents constructions for determinization and complementation of automata with various kinds of fairness constraints.

In our construction we don't do this. There is at no point a deterministic automaton that is equivalent to the starting automaton.

4 Determinization and Complementation

4.1 Previous Work

[SVW87] gives a procedure that complements an n state Büchi automaton into a non-deterministic Büchi automaton that can have as many as 2^{4n^2} states. In [Saf89], the complexity of translation between various forms of ω-automata is studied and procedures for these translations are presented. In the same work, the complexity of complementation for several kinds of automata is investigated, and constructions are given. Lower bounds are also provided on the complexities of these operations.

The following is a list of some relevant results for the determinization and complementation of ω-automata that are interesting for our language containment purposes. We use the same notation as [Saf88], i.e. N stands for non-deterministic, D stands for deterministic, B for Büchi, R for Rabin, and S for Streett automata. For instance, $NR(n,h) \longrightarrow NB((n+1)h)$ denotes the fact that a non-deterministic Rabin automaton with n states and h (L_i, U_i) pairs in its fairness condition can be converted to a non-deterministic Büchi automaton with $(n+1)h$ states.

1. $NB(n) \overset{\text{det}}{\rightarrow} DR(2^{O(n\ log\ n)}, n)$. This also yields a complementation procedure from NB into DS of the same complexity: only the fairness condition of the Rabin automaton needs to be complemented to achieve this. ([Saf89],[Saf88])
2. $NS(n, h) \overset{\text{det}}{\rightarrow} DR(2^{O(nh\ log\ nh)}, nh)$ ([Saf92])
3. $DR(n, h) \rightarrow DS(n2^{h\ log\ h}, h{+}1)$ and $DS(n, h) \rightarrow DR(n2^{h\ log\ h}, h{+}1)$ ([Saf89])
4. 1 and 3 imply that $NB(n) \overset{\text{det}}{\rightarrow} DS(2^{O(n\ log\ n)}, n)$. Using 1, a deterministic Rabin automaton is obtained, which is converted into a deterministic Streett automaton using 3.
5. 2 and 3 imply that $NS(n, h) \overset{\text{det}}{\rightarrow} DS(2^{O(nh\ log\ nh)}, nh)$ by a similar argument.
6. $NR(n, h) \rightarrow NB((n + 1)h)$.

[Cer92] addresses finite automata that are *k-step observably non-deterministic*, i.e., the selection of the next state can be identified by observing ahead the input string up to k symbols. In Section 4.4, we generalize this approach to work on ω-automata, and give a polynomial determinization construction.

We have implemented a “determinization and complementation tool-box” mainly based on ideas from the literature mentioned above. In the rest of this section, we discuss the algorithms that constitute the tool-box and their implementations.

The algorithms have been implemented as part of the Berkeley formal verification system HSIS [HSIS94]. In all of the following cases, a BDD representation of the non-deterministic automaton's transition relation is the input to the algorithms, and the output is a BDD representation for the transition relation of the determinized or complemented automaton.

4.2 Büchi Automata

As mentioned in [Saf88] the complexity of determinization for Büchi automata to Rabin automata is at least $2^{O(n\ log\ n)}$; strictly harder than the equivalent problem for finite automata.

Safra's Construction [Saf88] gives an asymptotically optimum construction for determinizing Büchi automata. The algorithm is provided here for reference. A proof of correctness can be found in [Saf88].

Given a non-deterministic Büchi automaton $N = \langle \Sigma, Q_N, I, \delta_N, F^\infty A\rangle$, a deterministic Rabin automaton $D = \langle \Sigma, Q_D, \{q_0\}, \delta_D, C\rangle$ accepting the same language is constructed as follows.

- **States (Q_D):** Each state of D is a labelled ordered tree. A labelled ordered tree is a rooted tree with the following attributes:
  - A complete ordering on the children of each node (i.e., if a and b are the children of the same node, either a is an “older sibling” of b or b is an “older sibling” of a)
  - A label on each node of the tree: Each node has some subset of Q_N as its label, with the following restrictions:
    - ∗ The union of the labels of the children of a node v is a proper subset of the label of v,

∗ Two nodes neither of which is an ancestor of the other must have disjoint sets as their labels.

Moreover, each node is colored *green* or *white*, and has a distinct *name*, which is an integer between 1 and $n = |Q_N|$. See Figure 4 for an example. A node a is said to be to the *left* of another node b if either a is older than b, or some ancestor of a is an older sibling of some ancestor of b.

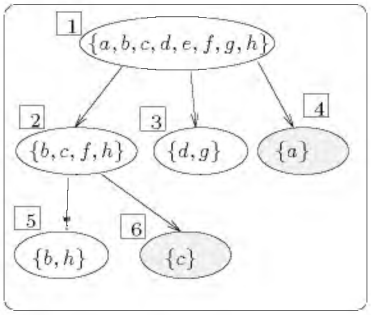


Fig. 4. One state of the deterministic automaton. In this example, $Q_N = \{a,b,c,d,e,f,g,h,k,m,s\}$. Green nodes are shaded.

- **The initial state** q_0 is a single node tree with the node colored white and having I, the set of initial states of N as its label. The name of the node is arbitrarily chosen to be 1.
- **The transition relation** δ_D**:** Given a state s of D and a symbol $\alpha \in \Sigma$, $\delta_D(s,\alpha)$ is computed by performing the following sequence of actions:
  1. Set the color of all the nodes of the labelled ordered tree s white.
  2. For every node of the tree s, labelled by the set $L \subseteq Q_N$, replace L with $\delta_N(L,\alpha)$.
  3. For every node v of s labelled L, if L contains accepting states (states from A), create a new node w to be the youngest child of v, and label it with $L \cap A$, i.e., the final states in the label of v.

     Steps (1-3) may result in a tree that does not satisfy the requirements stated before. The following steps restore the required structure.
  4. For every state q of N appearing on the label of a node v, if q appears on the label of a node that is to the "left" of v, remove q from the label of v.
  5. Remove all nodes with empty labels.
  6. For every node v whose label is equal to the union of the labels of its children, remove all descendants of v and color v green.
  7. If a node has been newly created, give it a name between 1 and n that is not the name of any other node on the tree. This is always possible, as the tree cannot have more than n nodes ([Saf88]).
  8. If a node has not been deleted (i.e., it appeared in s, and it got transformed but did not get deleted), leave its name the same.

The fairness condition is a Rabin type constraint given by $C = \bigvee_{1 \leq i \leq n}(F^{\infty} Gr_i \wedge G^{\infty} App_i)$ where Gr_i is the set of states of D where the node named i in the tree representing the state is colored green and App_i is the set of states of D where, in the tree representing the state, a node named i appears.

In [Saf88], it is proved that the deterministic Rabin automaton constructed consists of at most $2^{O(n\ log\ n)}$ states and n (L_i, U_i) pairs in its fairness constraint. Note that, since deterministic Rabin automata can be complemented into deterministic Streett automata in a straightforward manner, by rewriting the acceptance condition as $C = \bigwedge_{1 \leq i \leq n}(F^{\infty} \overline{App_i} \wedge G^{\infty} \overline{Gr_i})$ this construction provides a means for complementation with the same asymptotically optimum complexity of $2^{O(n\ log\ n)}$ [Saf88].

From here on not interesting anymore

Partly implicit implementation of Safra's Construction The construction in the previous section is one of the core routines in the tool-box. Safra's construction does not lend itself to a straightforward completely implicit implementation, since the information that the states encode does not provide us with a natural encoding. Moreover, even if such an encoding existed, it would have to encode $2^{O(n\ log\ n)}$ states, which requires at least $n\, log\, n$ binary variables. This is inefficient if n is large, no matter how simple D may be. We give a partly explicit implementation that is efficient if the determinized output automaton is relatively small. However, most of the computation of the transition relation is done implicitly. This point is made precise below.

Each state of the deterministic automaton is a labelled ordered tree, and can be identified by the information it encodes. We need a way of representing trees compactly and the representation needs to be canonical for computational efficiency. Binary Decision Diagrams (BDD's) not only come with these properties, but also are well suited for the computations required to transform a tree representing a state of the deterministic automaton to the one that represents the next state on a certain symbol. Multi-valued variables are encoded using binary ones. In our implementation a tree is represented as a relation, therefore by the BDD corresponding to its characteristic function. In the following, the term "relation" and the characteristic function of the relation will be used interchangeably.

- **The parenthood relation** $parent(node_1, node_2)$ is true if and only if $node_1$ is the (unique) parent of $node_2$ in the tree.
- **The color relation** $color(node_1, clr)$ is true if and only if $node_1$ in the tree is colored clr.
- **The seniority relation** $older(node_1, node_2)$ is true if and only if $node_1$ and $node_2$ have the same parent and $node_1$ is older than $node_2$ in the sense defined in Section 4.2.
- **The label relation** $label(node_1, state)$ is true if and only if $state$ is an element of the set of states that constitute the label of $node_1$.

A *node* is uniquely identified by its name, therefore, in the relations above, $node_1$ and $node_2$ have range $\{1, ..., n\}$. *state* ranges over Q_N, the set of states of N and

clr can take on values from $\{green, white\}$. The relation representing the tree is the collection of the relations above.

By representing trees with BDDs we expect several performance gains. First, most trees will have similar structures; therefore the parts of the information common to more than one tree will be represented only once. Although with this representation we cannot perform operations on sets of states, we can perform operations on sets of nodes in the tree, sets of input symbols, sets of labels, sets of colors, etc. This point is further detailed as the implementation of the algorithm is presented below. Another useful property of BDDs is their canonicity which provides a constant time routine for checking whether a state computed has been reached before.

The algorithm that constructs the transition relation for the deterministic Rabin automaton is an explicit breadth-first search on the state transition graph of the deterministic automaton. However, the states reached from a given state for all inputs $\alpha \in \Sigma$ are computed in one implicit step. Some of the interesting steps of this computation are explained below.

One step where the implicit representation comes in handy is the implementation of step 2 in Safra's algorithm. The implicit computation corresponding to this step is

$$next\_trees(input, select, node_1, node_2, state, color) =$$
$$\exists ps\ (\ tree(select, node_1, node_2, state \leftarrow ps, color) \wedge (ns \in \delta_N(ps, input)\ ))|_{ns \leftarrow state}$$

Here "$\exists\, ps$" denotes existential quantification of the variable ps, "$ns \leftarrow state$" denotes the substitution of the variable $state$ for the variable ns in the final BDD, and $input$ is a variable that takes on values from Σ. The output $next\_trees$ of this step carries the following information. If $input = \alpha$ is substituted in $next\_trees$, the result is a BDD which represents the tree with the new $label$ relation after step 2 of Safra's algorithm. Thus, the computation above yields the output of step 2 for all $\alpha \in \Sigma$, parametrized with respect to $input$. This is a case where a set of computations (those finding the new labels for each node for each input symbol) is performed in one step, as opposed to the $|\Sigma|$ iterations that would be required if the computation was performed sequentially. In hardware verification, typically $|\Sigma|$ is large, but the number of distinct next states of a given state is small. In such cases, our implementation using BDD's is very efficient.

Similarly, for step 4, the relations $older(node_1, node_2)$ and $parent(node_1, node_2)$ can be used in a fixed point computation to obtain $left(node_1, node_2)$, which consists of the pairs $(node_1, node_2)$ such that $node_1$ has an ancestor a_1 who is an older sibling of some a_2 who is an ancestor of $node_2$. Using this relation, all deletions in all the nodes of all next state trees can be made in one BDD operation, whereas the explicit implementation of step 4 would result in a traversal of each next state tree for each state of the non-deterministic automaton N.

As a final example, consider step 6 in Safra's algorithm. The relation expressing the set of nodes whose labels equal the union of those of their children can be computed by one intersection and existential quantification of the relations $parent(node_1, node_2)$ and $label(node_1, state)$ as

$$\begin{aligned}&union\_of\_children(node_1, state) = \\ &\qquad \exists node_2 \; (\; parent(node_1, node_2) \wedge label(node_1 \leftarrow node_2, state) \;) \\ &equal\_to\_children(node_1) = \\ &\qquad \forall state \; (\; union\_of\_children(node_1, state) \Leftrightarrow label(node_1, state) \;)\end{aligned}$$

whereas this would require traversing each node at least once in the explicit approach. Afterwards, all descendants of a set of nodes can be found by taking the transitive closure of the parenthood relation. Almost all other operations needed for finding the next states are expressible just as conveniently.

4.3 Streett and Rabin Automata

In our tool-box, Streett and Rabin automata are translated to non-deterministic Büchi automata first, and then the construction of Section 4.2 is used to determinize the resulting Büchi automata. The complexity of translating a Streett automaton into a Büchi automaton is known to be at least exponential in the number of (L_i, U_i) pairs ([Saf89], [SV89]), whereas for Rabin automata a polynomial translation exists ($NR(n,h) \rightarrow NB(n(h+1))$). For details of the translations, refer to [Taş95]. Most other kinds of frequently used ω-automata, such as L-processes and L-automata ([Kur87]) can be translated to Büchi automata efficiently.

4.4 Algorithms for Subclasses of Automata

Less complex algorithms can be used for determinization and complementation if further restrictions are imposed on the structure or the form of the fairness constraint of the automaton. Two interesting cases are examined in this section.

Automata with Fairness Constraint $G^\infty A$ In this case, a subset A of the states of the automaton are designated "good" and the run must eventually fall in A to be accepted.

We present a procedure for complementing an automaton $N = \langle \Sigma, Q_N, I, \delta_N, G^\infty A \rangle$ into a deterministic Büchi automaton $D = \langle \Sigma, Q_D, \{q_0\}, \delta_D, F^\infty C \rangle$. Since D is deterministic, complementing its fairness constraint yields an automaton accepting the same language as N, i.e., this construction also serves as a determinization procedure.

The intuition behind the construction is as follows. A string σ is in $\overline{L(N)}$ if and only if all runs of N on σ are not accepting, i.e., they all visit a state in $\overline{A} = Q_N - A$ infinitely often. We build a deterministic automaton D that keeps track of all runs of N, and checks whether all of them visit a state in $\overline{A}$ infinitely often. We achieve this by what can be viewed as a labelled subset construction. Every state of D is a set of states of N, each of them labelled "Good" or "Bad". From a state s of D on a symbol α, we go to state t which consists of all the states of N that are reachable from states in s on input α, and mark a state in t as "Good" if and only if either it is in A or all the states in s reaching it are labelled "Good" (we refer to this as propagating the labels). If all the states in s

are labelled "Good", s is made a final state of D and the labels on the states in s are not propagated. The unique run of a string σ in D visits final states infinitely often if and only if each run of σ in N visits a state in $\overline{A}$ infinitely often. A more formal specification of D and the proof of correctness of the construction follows.

- **States:** $Q_D = 2^{Q_N \times \{\text{Good,Bad}\}}$, i.e., subsets of states of N, with each state of N labelled "Good" or "Bad".
- **The initial state** $q_0 = I \times \{\text{Bad}\}$, i.e., the initial states of N labelled "Bad".
- **Accepting states** $C = 2^{Q_N \times \{\text{Good}\}}$, i.e., all states in Q_D which have all "Good" labels.
- **The transition relation** δ_D**:** For $s \in Q_D$, $\delta_D(s, \alpha)$ consists of pairs (q, l) such that
  - $q \in Q_N$ and $l \in \{\text{Good, Bad}\}$.
  - q is reachable from some state of N in s, i.e., $\exists(p, k) \in s$ such that $q \in \delta_N(p, \alpha)$.
  - If $s \notin C$, i.e., s is not an accepting state, then $(l = \text{Good})$ if and only if either q is in $\overline{A}$ or all states of N in s that reach q on α are labelled "Good". More formally, $(l = \text{Good})$ if and only if either $q \in \overline{A}$ or $\forall(p, k) \in s$ such that $q \in \delta_N(p, \alpha), k = \text{Good}$.
  - If $s \in C$, i.e. s is an accepting state having all "Good" labels, then $(l = \text{Good})$ if and only if q is in $\overline{A}$.

D has 2^{2n} states, some of them possibly unreachable. The proof of correctness of the construction hinges on the following lemma.

Lemma 1. *Let s be in C (i.e., a final state of D) and let t be reached from s on a string σ, with no final states visited along the way. A pair $(q, Good)$ is in t if and only if $q \in \overline{A}$ or all runs in N reaching from any state in q on string σ have visited a state in $\overline{A}$.*

Note that this construction can be generalized to fairness constraints of the form $\bigvee_{1 \leq i \leq h} G^\infty A_i$ by labelling each state of N in a state s of D "Good$_i$" or "Bad$_i$" for each A_i and changing the fairness constraint to a conjunction of the form $\bigwedge F^\infty \mathit{All\_Good}_i$, where $\mathit{All\_Good}_i$ consists of the states s of D in which all labels of states of N in s pertaining to A_i are "Good". This construction results in at most $2^{(h+1)n}$ states.

Our tool-box contains an implicit implementation of this construction.

k-step Observably Non-deterministic Automata In [Cer92] "k-step observably non-deterministic automata" (k-SOND) are studied. These automata are characterized by the fact that for any present state, the next state is uniquely determined by observing the input string up to k symbols ahead. A deterministic automaton corresponds to a 1-SOND automaton. The k-SOND property can be checked as follows. Let $T(ps, i, ns)$ be the transition relation of the automaton. Let $T^{(k)}$, the next state function with look-ahead of k, be given by

$$T^{(k)}(ps, i_1, i_2, ..., i_k, ns) = \exists ns_2 \exists ns_3 ... \exists ns_k \; [\, T(ps, i_1, ns) \wedge T(ns, i_2, ns_2) \wedge \\ T(ns_2, i_3, ns_3) ... T(ns_{j-1}, i_j, ns_j) ... T(ns_{k-1}, i_k, ns_k) \,]$$

Here $i_1, i_2, ..., i_k$ are variables with range Σ corresponding to the k following input symbols. The automaton is k-SOND if $T^{(k)}(ps, i_1, i_2, ..., i_k, ns)$ is deterministic, i.e., given ps and $i_1, i_2, ..., i_k$, the ns that yields $T^{(k)} = 1$ is uniquely determined. The determinacy check can be performed by computing the compatible projection of the BDD representing $T^{(k)}$ on the variable ns ([Lin91]) and verifying that the result is identical to $T^{(k)}$. Given fixed k, this procedure is polynomial in the size of the representation of T. If it is indeed the case that the automaton is k-SOND, complementation is trivial as a k-SOND automaton has at most one run on a given string. The automaton is first made complete by adding a dummy state and then the fairness constraint is complemented. For determinization we give the following construction, which is applicable to ω-automata with arbitrary fairness constraints.

Given a k-SOND automaton $N = \langle \Sigma, Q_N, \{q_0\}, \delta_N, \Phi \rangle$, we first construct an auxiliary automaton X that keeps record of the last k symbols in the input stream. The states of this automaton are $\Sigma^{<k}$, i.e., strings from Σ^* with length less than k. The initial state is the empty string ϵ, and X accepts all strings. Therefore, $X = \langle \Sigma, \Sigma^{<k}, \{\epsilon\}, \delta_X, True \rangle$, where, for $\alpha \in \Sigma$

- If $|\gamma| \geq k - 1$, then $\delta_X(\gamma, \alpha) =$ The last k symbols of $\gamma\alpha$.
- If $|\gamma| < k - 1$, then $\delta_X(\gamma, \alpha) = \gamma\alpha$.

We now construct the automaton N' which observes the state of X, i.e. its input alphabet is $\Sigma^{<k}$, the set of states of X. N' has the same state space as N except for an extra state q_{init}, i.e., $Q_{N'} = Q_N \cup \{q_{init}\}$. q_{init} is the initial state of N'. The transition relation of N', $\delta_{N'}(q, \gamma)$ is given as follows.

- If $|\gamma| < k$, then $\delta_{N'}(q, \gamma) = \{q\}$.
- If $|\gamma| = k$, then $\delta_{N'}(q, \gamma)$ consists of the unique next state of q in N if the next k symbols in the input stream were known to be given by γ.

The automaton N' is deterministic, by the definition of a k-SOND automaton. N' has the same fairness constraint as N. The product of X and N' is the deterministic automaton that is desired. $X \times N'$ is constructed as follows. The state space consists of pairs $(q_X, q_{N'})$ such that $q_X \in Q_X$ and $q_{N'} \in Q_{N'}$. The input alphabet is Σ. The transition relation is given as below.

$$\delta_{X \times N'}((q_X, q_{N'}), \alpha) = \{ \ (\ \delta_X(q_X, \alpha), \delta_{N'}(q_{N'}, q_X) \ ) \ \}$$

By construction, in the product $X \times N'$, on an input string σ the N' component of the product state traces with a delay of k steps the states that N would trace on the same input. Observe that, by the k-SOND property, N has at most one run on a given input string. Therefore, by imposing N's fairness constraint Φ on the N' component of the product state, we obtain an automaton that accepts the same language as N, and, by construction, is deterministic.

The number of states in $X \times N'$ is at most $|Q_{N'}|.|Q_X| = O(n|\Sigma|^{k+1})$, polynomial in n and $|\Sigma|$.

5 Experimental Results

We performed several experiments to gauge the applicability of our algorithms in connection with the language containment routines in HSIS. We must state, however, that, since no similar tools were available previously, we were able to find only a few industrial examples. The rest of the experiments were performed on artificial examples which were put together to demonstrate various aspects of the implementations.

For property verification, we worked on properties of the form "if event e happens, then one of the acceptable responses in S must eventually follow" and "the resource is accessed in a fair fashion by all the units that demand it". For these properties, the input and output automata are typically small (hundreds of states) and our tools handle them easily.

The more challenging application domain for our algorithms is hierarchical verification. We believe this is of more practical interest in the digital hardware design community. We studied three examples.

- A bit data link controller, which was part of an actual CPU design, had been abstracted by hand because it was too large for the verification tool to handle. The original design had ≈ 50 registers. Parts of the hardware were removed, and the rest of the transitions that depend on signals from the abstracted part were made non-deterministic, so that the abstract version of the design had ≈ 20 registers. Language containment check needed to be performed to show the validity of this abstraction, which could not be done at the time because the verifier they used did not support language containment between non-deterministic automata. Later in the process, the designers found an error in the abstraction. With our tools, on the other hand, the abstracted non-deterministic module was easily complemented to yield an automaton with less than 100 states.
- A distributed memory multi-processor cache consistency protocol. This was an artificial example. We specified the protocol using non-deterministic automata to represent the desired behavior of the processors and an arbiter. We proved certain properties involving cache consistency using these automata. We then constructed more detailed, deterministic implementations for the modules and, using our tool-box, proved that each of them implements the protocol specification correctly. In this case, the output automata had less than 1000 states.
- The environment of a terminal was modelled using a non-deterministic automaton. (An example from industry.) To check if this abstraction was correct, we applied our tools to complement the environment, which had ≈ 20 state variables and obtained an output automaton with less than 100 states.

To examine the limits of our current implementation, we took a scalable example and applied our partly implicit determinization algorithm to it. We were able to go to approximately 100,000 states in the determinized automaton on a DECstation 5000/260 with 480 MB of memory. This number is much larger than that needed for the applications we ran our algorithm on.

The availability of both implicit and explicit algorithms in the tool-box offers flexibility on the kinds of examples on which the tool-box can be applied. To demonstrate this, on a set of automata with fairness constraints of the form $G^{\infty} A$, we compared the performance of the implicit algorithm we gave for this special case (denoted by $\mathcal{IM}$ in this paragraph) with that of the partly explicit algorithm (denoted by $\mathcal{EX}$). On a set of scalable examples, we compared run time and memory consumption. While $\mathcal{EX}$ used resources linear in the size of the determinized automaton, $\mathcal{IM}$'s performance showed little dependence on this. $\mathcal{IM}$'s resource usage depended more directly on the input automaton, which determines the number of state variables needed to encode the state of the determinized automaton. For a set of input automata of fixed size n, by k we denote the size of the determinized automaton. For small k, $\mathcal{EX}$ has an advantage. As k increases, the resource usage of $\mathcal{IM}$ remains almost constant, while that of $\mathcal{EX}$ increases linearly. Thus, after a crossover point, $\mathcal{IM}$ was more advantageous and handled cases that $\mathcal{EX}$ could not handle. (In one such case, the output automaton had more than 2^{16} states.) It needs to be noted, however, that $\mathcal{EX}$ is the only feasible alternative if the input automaton is large but the output automaton is still of reasonable size.

Although the memory consumption of the partly explicit algorithm is the limiting factor for larger examples, this does not affect further stages of the verification tool. It can be shown that ([Taş95]) the BDD representing the transition relation of the output automaton has at most $m.|\Sigma|.(2.\lceil log\, m\rceil + \lceil log\, |\Sigma|\rceil)$ nodes. Since the amount of memory used by the algorithm to represent each state and transition is much more than that needed for one BDD node, the BDD representing the transition relation takes up much less memory than the algorithm itself uses.

There is room for further improvement in our implementation. Our main goal was to show that determinization algorithms, and therefore, language containment check between non-deterministic automata can be done in practice in many cases.

6 Conclusion

We have studied non-deterministic automata, and presented algorithms and techniques that make their use in verification systems possible. We have demonstrated the usefulness of non-determinism, and investigated its applications in computer-aided verification. The main contribution of our work is a set of tools that can determinize and complement widely used kinds of ω-automata. We have also addressed simpler cases where less complex techniques can be employed. By testing our tools on actual examples, we have shown that in many cases of practical interest it is computationally feasible to use determinization and complementation algorithms as part of a verification system.

One limitation of our current implementation is the need to convert to a Büchi automaton from a Streett automaton before determinization. In [Saf92] a technique for determinizing a Streett automaton ($DS(n, h)$) directly into a Rabin automaton ($DR(2^{O(nh\, log\, nh)}, nh)$) is given, while our approach would yield $2^{O(n2^h\, log\, (n2^h))}$ states in the worst case. This is a largely theoretical point; if the worst case behavior is observed, both approaches are far from practical. However, whether it is the case in practice that [Saf92]'s approach handles a large number of fairness constraints better than our approach remains to be investigated.

Checking for simulation relations between two non-deterministic automata can be viewed as a conservative approximation to checking language containment. This check is polynomial if there are no fairness constraints, but is NP-complete for a weak notion of fair simulation relations for Streett conditions ([DHW91],[HB95]). The most intuitive way to refine an abstract design is to replace a state with a set of states that add more detail, in which case a simulation relation is likely to exist. The shortcomings of this approach are that language containment can hold while no simulation relation exists, and that, in case no simulation relations are found, it is difficult to get feedback information to guide the modification of the design so as to make the check pass. The performances and practical applicability of checking for simulation relations and our approach need to be compared.

References

[Cho74] Y. Choueka. Theories of Automata on ω-Tapes: A Simplified Approach. *Journal of Computer and System Sciences*, 8:117-141, 1974.

[SVW87] A.P. Sistla, M.Y. Vardi and P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217-237, 1987.

[Kur87] R. P. Kurshan. Reducibility in Analysis of Coordination. In *Discrete Events Systems: Models and Applications*, volume 103 of LINCS, pages 19-39, 1987

[Lin91] B. Lin. Synthesis of VLSI Designs with Symbolic Techniques. *Ph.D. thesis, University of California, Berkeley*, 1991.

[Saf89] S. Safra. Complexity of Automata on Infinite Objects. *Ph.D. thesis, The Weizman Institute of Science, Rehovot, Israel*, March 1989.

[Saf88] S. Safra. On the Complexity of ω-Automata. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 319-327, 1988.

[SV89] S. Safra, and M.Y.Vardi On ω-Automata and Temporal Logic. In *Proc. 21st ACM Symp. on Theory of Computing*, 1989.

[Kur92] R. P. Kurshan Automata-Theoretic Verification of Coordinating Processes. UC Berkeley Lecture Notes, 1992.

[Saf92] S. Safra. Exponential Determinization for ω-Automata with Strong-Fairness Acceptance Conditions. In *Proc. 24th ACM Symposium on Theory of Computing*, 1992.

[Saf93] S. Safra. Private Communication.

[Cer92] E. Cerny. Verification of I/O Trace Set Inclusion for a Class of Non-deterministic Finite State Machines. In *Proc. Int'l Conference on Computer Design*, Cambridge, pages 526-529, October 1992.

[DHW91] D. L. Dill, A. J. Hu and H. Wong-Toi Checking for Language Inclusion Using Simulation Preorders. In Proc. of the Third Workshop on Computer-Aided Verification, 1991.

[HB95] R. Hojati and R. K. Brayton Computing Fair Simulation Relations. Unpublished manuscript.

[HSIS94] A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan A. L. Sangiovanni-Vincentelli T. R. Shiple V. Singhal, S. Taşıran, H.-Y. Wang. HSIS: A BDD-Based Environment for Formal Verification. In Proc. of the Design Automation Conf., June 1994.

[Taş95] S. Taşıran Language Containment Using Non-deterministic ω-Automata. *M.S. thesis, University of California, Berkeley*, 1995.

Appendix: Proofs

Proof (Lemma 1). By induction on the length of σ. For $|\sigma| = 0$ or 1 the statement holds by the construction of the next state function. Assume that the statement holds for $|\sigma| = k$. For$|\sigma| = k + 1$, let $\sigma = \gamma\alpha$, where $|\gamma| = \alpha$. Let r be the state of D reached from s on γ. A state q in t is labelled as "Good" if and only if all of its predecessors in r (all states of N that reach q on α) are labelled "Good", or q is in A. By the induction assumption, the states of N in r are labelled "Good" if and only if all runs that reached them from states in s on γ have visited a state in $\overline{A}$. Therefore, a state q in t is labelled as "Good" if and only if all runs reaching it have visited a state in $\overline{A}$ or q itself is in $\overline{A}$.

Lemma 2. *The construction for complementing G^{∞} automata given in Section 4 is correct, i.e., a string σ is accepted by D if and only if it is not accepted by N.*

Proof (Lemma 2, $\Rightarrow$ direction). Assume that σ is accepted by D. Then the run of D on σ visits states in C infinitely often. Let $c_0, c_1, c_2, ...$ be the sequence of final states of D visited along the run, and let σ_i denote the part of σ that takes us from c_i to c_{i+1}. Let m be any run of N on σ. Let m_0, m_1, m_2 be the states on m that fall into $c_0, c_1, c_2, ...$, respectively. All th m_i's must be labelled "Good". By Lemma 1, an m_i is labelled "Good" if all runs in N on x_{i-1} from all states in c_{i-1} (therefore m_{i-1} too) go through a state in $\overline{A}$. Therefore, m visits states in $\overline{A}$ infinitely often. Since $|\overline{A}|$ is finite, m visits some state in $\overline{A}$ infinitely often. Thus, $inf(m) \nsubseteq A$ for any run m on σ. This implies that $G^{\infty}A$ does not hold for any run.

Proof (Lemma 2, $\Leftarrow$ direction). Assume towards a contradiction that σ is accepted by both N and D. Then there exists a run m of N on σ that eventually visits states in A only. Also, since D accepts σ, the run of D on σ goes through an infinite sequence of states in C, $c_0, c_1, c_2, ...$ Let c_j be a state in C visited during the infinitary part of the run in D. From c_j to c_{j+1}, there is a run in N from a state in c_j (the one that is on m) to a state in c_{j+1} (again the one on m) that goes through states in A only. This contradicts Lemma 1.

This article was processed using the LaTeX macro package with LLNCS style