Searching Powerset Automata by Combining Explicit-State and Symbolic Model Checking

Alessandro Cimatti[1], Marco Roveri[1,2], and Piergiorgio Bertoli[1]

[1] ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy
Phone: +39 0461 314517, Fax: +39 0461 413591.
`{cimatti,roveri,bertoli}@irst.itc.it`
[2] DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract. The ability to analyze a digital system under conditions of uncertainty is important in several application domains. The problem is naturally described in terms of search in the powerset of the automaton representing the system. However, the associated exponential blow-up prevents the application of traditional model checking techniques. This work describes a new approach to searching powerset automata, which does not require the explicit powerset construction. We present an efficient representation of the search space based on the combination of symbolic and explicit-state model checking techniques. We describe several search algorithms, based on two different, complementary search paradigms, and we experimentally evaluate the approach.

Keywords: Explicit-State Model Checking, Symbolic Model Checking, Binary Decision Diagrams, Synchronization Sequences

1 Introduction

The ability of analyzing digital systems under conditions of uncertainty is extremely useful in various application domains. For hardware circuits, it is important to be able to determine homing, synchronization and distinguishing sequences, which allow to identify the status of a set of circuit flip-flops. For instance, synchronization sequences, i.e. sequences that will take a circuit from an unknown state into a completely defined one [9], are used in test design and equivalence checking. Similar problems are encountered in automated test generation, e.g. to determine what sequence of inputs can take the (black-box) system under test in a known state. In Artificial Intelligence, reasoning with uncertainty has been recognized as a significant problem since the early days. For instance, the Blind Robot problem [11] requires to plan the activity for a sensorless agent, positioned in any location of a given room, so that it will be guaranteed to achieve a given objective.

Such problems are naturally formulated as search in the powerset of the space of the analyzed system [9]: a certain condition of uncertainty is represented as the set of indistinguishable system states. However, search in the powerset space yields an exponential blow-up. A straightforward application of symbolic model checking techniques is hardly viable: the symbolic representation of the powerset automaton requires exponentially more variables than needed for the analyzed system. On the other hand, approaches based on explicit-state search methods tend to suffer from the enumerative nature of the algorithms.

In this work, we propose a new approach to the problem of searching the powerset of a given nondeterministic automaton which does not require the explicit powerset automaton construction. The approach can be seen as expanding the relevant portion of the state space of the powerset automaton on demand, and allows to tackle in practice rather complex problems. The approach combines techniques from symbolic model checking, based on the use of Binary Decision Diagrams (BDDs) [3], with techniques from explicit-state model checking. We represent in a fully symbolic way sets of sets of states, and we provide for the efficient manipulation of such data structures. Using this representation, we tackle the problem of finding an input sequence which guarantees that only states in a target set will be reached for all runs, regardless of the uncertainty on the initial condition and on nondeterministic machine behaviors. We present several algorithms based on two different search paradigms. The *fully-symbolic* paradigm allows to perform a breadth-first search by representing the frontier as a single symbolic structure. In the *semi-symbolic* paradigm, search is performed in the style of explicit-state model checking, considering at each step only a (symbolically represented) element of the search space, i.e. a set of states. Both search paradigms are based on fully symbolic primitives for the expansion of the search space, thus overcoming the drawbacks of an enumerative approach.

The algorithms return with failure if and only if the problem admits no solution, otherwise a solution is returned. Depending on the style of the search, the solution can be guaranteed to be of minimal length. We also present an experimental evaluation of our algorithms, showing that the paradigms are complementary and allow to tackle quite complex problems efficiently.

The paper is structured as follows. In Section 2 we introduce the problem. In Section 3 we describe the techniques for the implicit representation of the search space, and in Section 4 we present the semi-symbolic and fully-symbolic search paradigms. In Section 5 we present an experimental evaluation of our approach. In Section 6 we discuss some related work and draw the conclusions.

2 Intuitions and Background

We consider nondeterministic finite state machines. $\mathcal{S}$ and $\mathcal{A}$ are the (finite) sets of states and inputs of the machine. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation. We use s and s' to denote states of $\mathcal{S}$, and α to denote input values. In the following, we assume that a machine is given in the standard BDD-based representation used in symbolic model checking [10]. We call $\boldsymbol{x}$ and $\boldsymbol{x}'$ the vectors of current and next state variables, respectively, while $\boldsymbol{\alpha}$ is the vector of input variables. We write $\boldsymbol{\alpha} = \alpha$ for the BDD in the $\boldsymbol{\alpha}$ variables representing the input value α. When clear from the context, we confuse the set-theoretic and symbolic

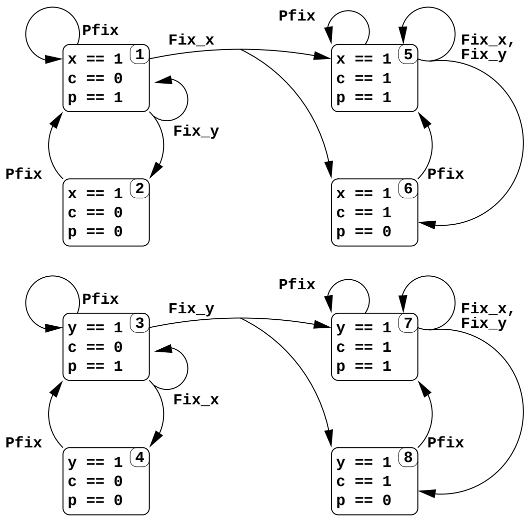

Fig. 1. The example automaton

representations. For instance, we use equivalently the *False* BDD and $\emptyset$. We write $\mathcal{R}(\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{x}')$ for the BDD representing the transition relation to stress the dependency on BDD variables. We say that an input α is acceptable in s iff there is at least a state s' such that $\mathcal{R}(s,\alpha,s')$ holds. The acceptability relation is represented symbolically by $\textsc{Acc}(\boldsymbol{x},\boldsymbol{\alpha}) \doteq \exists \boldsymbol{x}'.\mathcal{R}(\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{x}')$. An input sequence is an element of $\mathcal{A}^*$. We use ϵ for the 0-length input sequence, π and ρ to denote input sequences, and $\pi;\rho$ for concatenation.

In this paper we tackle the problem of finding an input sequence that, if applied to the machine from any initial state in $\mathcal{I} \subseteq \mathcal{S}$, guarantees that the machine will reach a target set of states $\mathcal{G} \subseteq \mathcal{S}$, regardless of nondeterminism. We use for explanatory purposes the simple system depicted in figure 1. A circuit is composed of two devices, x and y. The circuit is malfunctioning ($c = 0$), and the reason is that exactly one of the devices is faulty (i.e. $x = 0$ or $y = 0$). It is possible to fix either device (input Fix_x and Fix_y), but only if a certain precondition p is met. Fixing the faulty device has the effect of fixing the circuit ($c = 1$), while fixing the other one does not. Fixing either device has the uncertain effect of spoiling the fixing precondition condition (i.e. $p = 0$). $Pfix$ has the effect of restoring the fixing precondition ($p = 1$). Each state is given a number, and contains all the propositions holding in that state. For instance, state 1 represents the state where device x is the reason for the fault, and fixing is possible. Given that only one device is faulty, $x = 0$ also stands for $y = 1$, and vice versa.

The problem is finding an input sequence which fixes the circuit, taking the machine from any state in $\mathcal{I} = \{1, 2, 3, 4\}$ (where the circuit is faulty, but we don't know if the reason is in device x or y, nor if fixing is possible) to the target set $\mathcal{G} = \{5, 7\}$ (the circuit is fixed, and the fixing condition is restored).

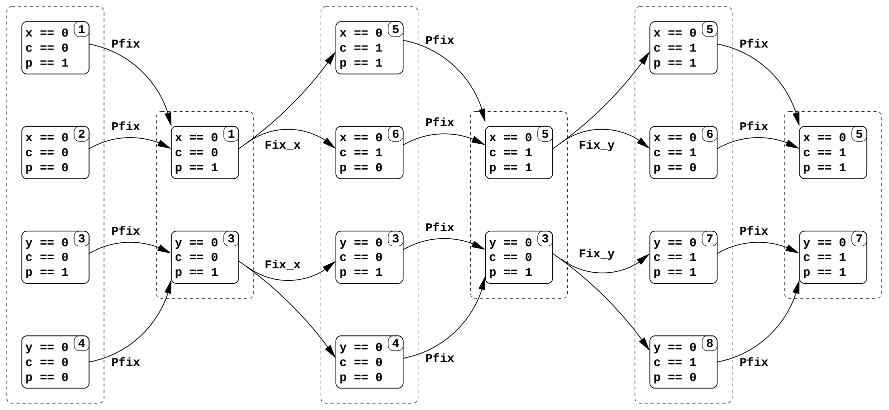

Fig. 2. A solution to for example problem

A possible solution is the input sequence: $Pfix$; Fix_x ; $Pfix$; Fix_y; $Pfix$. Figure 2 shows why this is the case. The initial uncertainty is in that the system might be in any of the states in $\{1, 2, 3, 4\}$. This set is represented in figure 2 by a dashed line. We call such a set an *uncertainty state* as in [2]. Intuitively, an uncertainty state expresses a condition of uncertainty about the system, by collecting together all the states which are indistinguishable while analyzing the system. An uncertainty state is an element of $\text{Pow}(\mathcal{S})$, i.e. the powerset of the set of states of the machine. The first input value, $Pfix$, makes sure that fixing is possible. This reduces the uncertainty to the uncertainty state $\{1, 3\}$. Despite the remaining uncertainty (i.e. it is still not known which component is responsible for the circuit fault), the following input value Fix_x is now guaranteed to be acceptable because it is acceptable in both states 1 and 3. Fix_x has the effect of removing the fault if it depends on device x, and can nondeterministically remove the precondition for further fixing ($p = 0$). The resulting uncertainty state is $\{3, 4, 5, 6\}$. The following input, $Pfix$, restores $p = 1$, reducing the uncertainty to the uncertainty state $\{3, 5\}$, and guarantees the acceptability of Fix_y. After Fix_y, the circuit is guaranteed to be fixed, but p might be 0 again (states 6 and 8 in the uncertainty state $\{5, 6, 7, 8\}$). The final $Pfix$ reduces the uncertainty to the uncertainty state $\{5, 7\}$, and guarantees that only target states are reached.

The following definition captures the intuitive arguments given above.

Definition 1. *An input α is acceptable in an uncertainty state $\emptyset \neq Us \subseteq \mathcal{S}$ iff α is acceptable in every state in Us, i.e. $\exists \boldsymbol{\alpha}.\forall \boldsymbol{x}.((Us(\boldsymbol{x}) \wedge \boldsymbol{\alpha} = \alpha) \rightarrow \textsc{Acc}(\boldsymbol{x}, \boldsymbol{\alpha}))$ is not $\emptyset$.*

If α is acceptable in Us, its image $Image[\alpha](Us)$ is the set of all the states reachable from Us under α, i.e. $\exists \boldsymbol{x}.(Us(\boldsymbol{x}) \wedge \exists \boldsymbol{\alpha}.(\boldsymbol{\alpha} = \alpha \wedge \mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')))[\boldsymbol{x}'/\boldsymbol{x}]$ where $[\boldsymbol{x}'/\boldsymbol{x}]$ represents parallel variable substitution.

The image of an input sequence π in an uncertainty state, written $Image[\pi](Us)$, is defined as follows.

$$\begin{array}{lcl} Image[\epsilon](Us) & \doteq & Us \\ Image[\pi](\emptyset) & \doteq & \emptyset \\ Image[\alpha;\pi](Us) & \doteq & \emptyset, \ \textit{if } \alpha \textit{ is not acceptable in } Us \\ Image[\alpha;\pi](Us) & \doteq & Image[\pi](Image[\alpha](Us)), \ \textit{otherwise} \end{array}$$

The input sequence π is a solution to the powerset reachability problem from $\emptyset \neq \mathcal{I} \subseteq \mathcal{S}$ to $\emptyset \neq \mathcal{G} \subseteq \mathcal{S}$ iff $\emptyset \neq Image[\pi](\mathcal{I}) \subseteq \mathcal{G}$.

Search in Pow($\mathcal{S}$) can be performed either forwards (from the initial uncertainty state $\mathcal{I}$ towards the target uncertainty state $\mathcal{G}$) or backwards (from $\mathcal{G}$ towards $\mathcal{I}$). Figure 2 depicts a subset of the search space when proceeding forwards. The full picture can be obtained by considering the effect of the other input values to the uncertainty states. For instance, the input values $Pfix$ on the second uncertainty state $\{1, 3\}$ would result in a self loop, while Fix_y would lead to $\{1, 3, 7, 8\}$. The first and third uncertainty states can not be expanded further, because the input values Fix_x and Fix_y are not acceptable. When a nonempty uncertainty state $Us_i \subseteq \mathcal{G}$ is built from $\mathcal{I}$, the associated input sequence (labeling a path from $\mathcal{I}$ to Us_i) is a solution to the problem.

Figure 3 depicts the backward search space. The levels are built from the target states, on the right, towards the initial ones, on the left. At level 0 we have the pair $\langle\{5, 7\}\ .\ \epsilon\rangle$, composed of an uncertainty state and an input sequence. We call such a pair uncertainty state-input sequence (UsS) pair. The dashed arrows represent the *strong preimage* of each Us_i under the input value α_i, i.e. the extraction of the maximal uncertainty state where the α_i is acceptable, and guaranteed to result into the uncertainty state being expanded. At level 1, only the UsS pair $\langle\{5, 6, 7, 8\}\ .\ Pfix\rangle$ is built, since the strong preimage of the uncertainty state 0 for the inputs Fix_x and Fix_y is empty. At level 2, there are three UsS pairs, with (overlapping) uncertainty states Us2, Us3 and Us4, associated, respectively, with the length 2 sequences $Fix_x; Pfix$, $Pfix; Pfix$ and $Fix_y; Pfix$. (While proceeding backwards, a sequence is associated with an uncertainty state Us*i* if it labels a path from Us*i* to the target set.) Notice that Us3 is equal to Us1, and therefore deserves no further expansion. The expansion of uncertainty states 2 and 4 gives the uncertainty states 5 and 6, both obtained by the strong preimage under $Pfix$, while the strong preimage under inputs Fix_x and Fix_y returns empty uncertainty states. The further expansion of Us5 results in three uncertainty states. The one resulting from the strong preimage under $Pfix$ is not reported, as equal to Us5. Uncertainty state 7 is also equal to Us2, and deserves no further expansion. Uncertainty state 8 can be obtained by expanding both Us5 and Us6. At level 5, the expansion produces Us10, which contains all the initial states. Therefore, both the corresponding sequences are solutions to the problem.

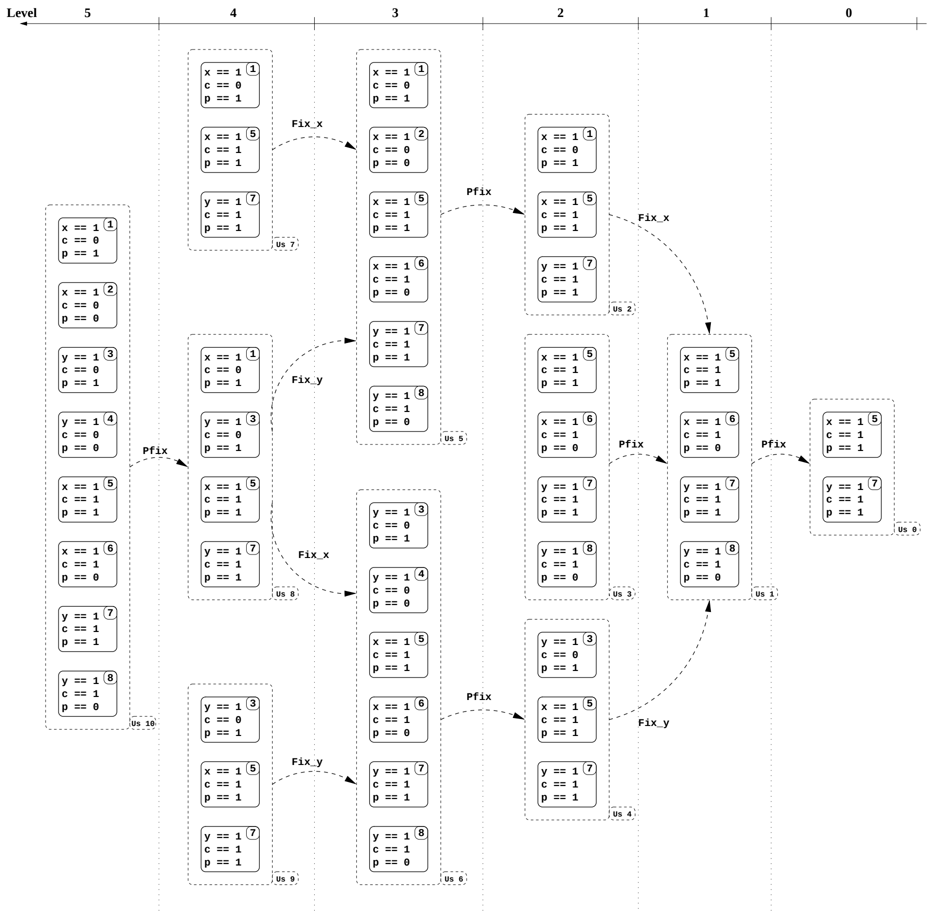

Fig. 3. The Search Space for the Example Problem

3 Efficient Representation of Pow($\mathcal{S}$)

In this section we describe the symbolic representation of the search space Pow($\mathcal{S}$), and the primitives used in the search. Our representation mechanisms combines elements used in symbolic and explicit-state model checking. The first ingredient is a standard BDD package, providing for the symbolic representation mechanism. Each uncertainty state Us is directly represented by the BDD $Us(\boldsymbol{x})$, whose models are exactly the states contained in Us. In practice, the uncertainty state is the pointer to the corresponding BDD. The second ingredient, from explicit-state model checking, is a hash table, which is used to store and retrieve pointers to the (BDDs representing the) uncertainty states which have been visited during the search. The approach heavily relies on the normal form of BDD, which allow for comparison in constant time. Figure 4 gives

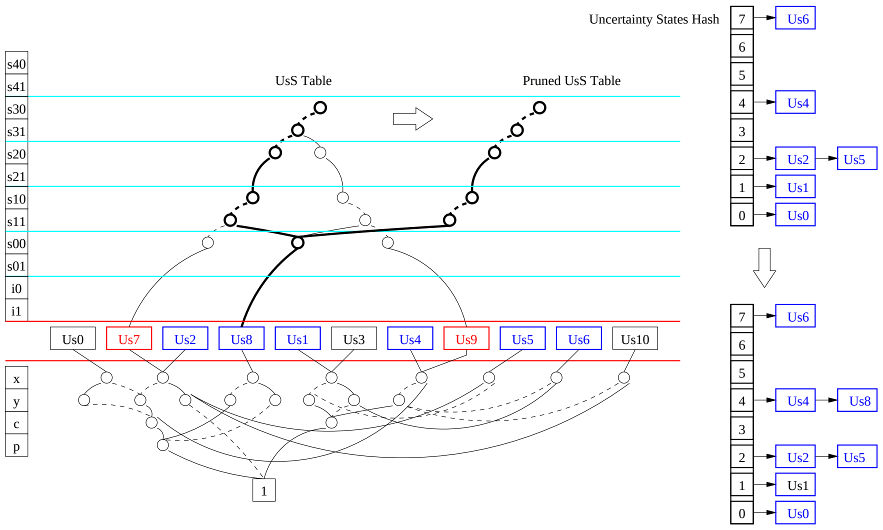

Fig. 4. The combined use of BDD and the cache

overview of the approach on the data structure built while analyzing the example. The column on the left shows the variables in the BDD package. Let us focus first on the lower part that contains the state variables, i.e. x, y, c and p. (The upper variables, including the input variables, will be clarified later in this section.) Each uncertainty state in figure 3 is represented by a (suitably labeled) BDD, shown in the picture as a subgraph. (Solid [dashed, respectively] arcs in a BDD represent the positive [negative, resp.] assignment to the variable in the originating node. For the sake of clarity, only the paths leading to $True$ are shown.) On the right hand side, two configurations of the visited uncertainty states hash are shown. The picture gives an example of the potential memory savings which can be obtained thanks to the great ability of the BDD package to minimize BDD memory occupation. Besides the uniqueness, there is a large amount of sharing among different BDDs: for instance, Us6 and Us10 share their sub-nodes with the previously constructed Us2, Us3 and Us4. Furthermore, the set-theoretic operations for the transformation and combination of uncertainty states (e.g. projection, equivalence, inclusion) can be efficiently performed with the primitives provided by the BDD package. The advantage over an enumerative representation of uncertainty states (e.g. the list of the state vectors associated to each state contained in the Us) is evident.

The exploration of Pow($\mathcal{S}$) is based on the use of UsS tables, i.e. sets of UsS pairs, of the form $UsST = \{\langle Us_1 \ . \ \pi_1\rangle, \ldots, \langle Us_n \ . \ \pi_n\rangle\}$ where the π_i are input sequences of the same length, such that $\pi_i \neq \pi_j$ for all $1 \leq j \neq i \leq n$. We call Us_i the uncertainty set indexed by π_i. When no ambiguity arises, we write $UsST(\pi_i)$ for Us_i. A UsS table allows to represent a level in the search space. For instance, when proceeding backward (see figure 3), each UsS pair $\langle Us_i \ . \ \pi_i \rangle$ in the UsS table is such that Us_i is the maximal uncertainty state in which the associated input sequence is acceptable, and its image is contained in the target states. When proceeding forward, for each UsS pair, the uncertainty state is the result of the application of the corresponding input sequence to the initial set.

The key to the efficient search is the symbolic representation of UsS tables, which allows for compactly storing sets of sets of states (annotated by input sequences) and their transformations. A UsS table $\{ \ \langle \{s_1^1, \ldots, s_{n_1}^1\} \ . \ \pi_1 \rangle, \ \ldots, \ \langle \{s_1^k, \ldots, s_{n_k}^k\} \ . \ \pi_k \rangle \ \}$ is represented as a relation between input sequences of the same length and states, by associating directly to each state in the uncertainty state the indexing input sequence, i.e. $\{ \ \langle s_1^1 \ . \ \pi_1 \rangle, \ \ldots, \langle s_{n_1}^1 \ . \ \pi_1 \rangle, \ \ldots, \ \langle s_1^k \ . \ \pi_k \rangle, \ \ldots, \langle s_{n_k}^k \ . \ \pi_k \rangle \ \}$. Given this view, the expansion can be obtained symbolically as follows. Let us consider first the UsS table $\{\langle Us \ . \ \epsilon \rangle\}$ represented by the BDD $Us(\boldsymbol{x})$. The backward step of expansion BwdExpandUsSTable constructs the UsS table containing the strong preimage of Us under each of the input values. This is the set of all state-input pairs where the input is acceptable in the state and all the successor states are in Us. Symbolically, we compute

$$\forall \boldsymbol{x}'.(\mathcal{R}(\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{x}') \rightarrow Us(\boldsymbol{x})[\boldsymbol{x}/\boldsymbol{x}']) \ \wedge \ \textsc{Acc}(\boldsymbol{x},\boldsymbol{\alpha})$$

i.e. a BDD in the $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ variables. This represents a relation between states and length-one input sequences, i.e. a UsS table where each Us_i is annotated by a length-one input sequence α_i.

The dual forward step FwdExpandUsSTable expands $\{\langle Us \ . \ \epsilon \rangle\}$ by computing the images of Us under every acceptable input:

$$(\exists \boldsymbol{x}.(Us(\boldsymbol{x}) \wedge (\forall \boldsymbol{x}.(Us(\boldsymbol{x}) \rightarrow \textsc{Acc}(\boldsymbol{x},\boldsymbol{\alpha})) \wedge \mathcal{R}(\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{x}'))))[\boldsymbol{x}/\boldsymbol{x}']$$

The resulting BDD represents a UsS table, where each Us_i is annotated by a length-one input sequence α_i such that $\emptyset \neq \mathit{Image}[\alpha_i](Us) = Us_i$.

In the general case, a UsS tables can contain longer input sequences, and the vector $\boldsymbol{\alpha}$ of input variables is not enough. Therefore, we use additional variables to represent the values of the input sequence at the different steps. To represent input sequences of length i, we need i vectors of new BDD variables, called *sequence variables*. The vector of sequence variables representing the i-th value of the sequence is written $\boldsymbol{\pi}_{[\mathbf{i}]}$, with $|\boldsymbol{\pi}_{[\mathbf{i}]}| = |\boldsymbol{\alpha}|$. Figure 4 shows the UsS table representing the third level of backward search space depicted in figure 3. The upper variables in the order are the input variables $i0$ and $i1$ and the sequence variables. When searching forwards [backwards, respectively] $\boldsymbol{\pi}_{[\mathbf{i}]}$ is used to encode the i-th [i-th to last, resp.] value in the sequence. The backward expansion primitive BwdExpandUsSTable can be applied in the general case to a UsS table $\mathit{UsST}_{i-1}(\boldsymbol{x},\boldsymbol{\pi}_{[\mathbf{i-1}]},\ldots,\boldsymbol{\pi}_{[\mathbf{1}]})$, associating an uncertainty state to plans of length $i-1$:

$$(\forall \boldsymbol{x}'.(\mathcal{R}(\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{x}') \rightarrow \mathit{UsST}_{i-1}(\boldsymbol{x},\boldsymbol{\pi}_{[\mathbf{i-1}]},\ldots,\boldsymbol{\pi}_{[\mathbf{1}]})[\boldsymbol{x}/\boldsymbol{x}']) \wedge \textsc{Acc}(\boldsymbol{x},\boldsymbol{\alpha}))[\boldsymbol{\alpha}/\boldsymbol{\pi}_{[\mathbf{i}]}]$$

As in the length-one case, the next state variables $\boldsymbol{x}'$ in $\mathcal{R}$ and in UsST_{i-1} (resulting from the substitution) disappear because of the universal quantification.

The input variables $\boldsymbol{\alpha}$ are renamed to the newly introduced plan variables $\boldsymbol{\pi}_{[\mathbf{i}]}$, so that in the next step of the algorithm the construction can be repeated. The forward step is defined dually. Notice that the fully symbolic expansion of UsS tables avoids the explicit enumeration of input values. This can lead to significant advantages when only a few distinct uncertainty states result from the application of all possible input values.

For either search directions, every time a UsS table is built its uncertainty states have to be compared with the previously visited uncertainty states. If not present, they must be inserted in the hash of the visited uncertainty states, otherwise eliminated. This analysis is performed by a special purpose primitive, called PruneUsSTable, which operates directly on the BDD representing the UsS table. The primitive assumes that in the BDD package input and sequence variables precede state variables (see figure 4). PruneUsSTable recursively descends the UsS table, and interprets as an uncertainty state every BDD node having a state variable at its top. It accesses the hash table of the previously visited uncertainty states with the newly found Us: if it is not present, then it is stored and returned, otherwise $False$ BDD is returned, and the traversal continues on different branches of the input and sequence variables. In this way, a new UsS table is built, where only the Us which had not been previously encountered are left. The pruning step also takes care of another source of redundancy: UsS tables often contain a large number of equivalent input sequences, all indexing exactly the same uncertainty state (in figure 3, two equivalent input sequences are associated with Us8). The resulting UsS table is such that, for each Us, only one (partial) assignments to the input and sequence variables is left. This simplification can sometime lead to dramatic savings.

4 Algorithms for Searching Pow($\boldsymbol{\mathcal{S}}$)

In this section we present two examples of search algorithms based on the data structures and primitives described in previous section. Both algorithms take in input the problem description in form of the BDDs $\mathcal{I}(\boldsymbol{x})$ and $\mathcal{G}(\boldsymbol{x})$, while the transition relation $\mathcal{R}$ is assumed to be globally available.

Figure 5 presents the *semi-symbolic forward* search algorithm. The algorithm represents the input sequences associated with the (symbolically represented) uncertainty states visited during the search as (explicit) lists of input values. The algorithm is based on the expansion of individual uncertainty states. OpenUsPool contains the (annotated) uncertainty states which have been reached but still have to be explored, and is initialized to the first uncertainty state of the search, i.e. $\mathcal{I}$, annotated with the empty input sequence ϵ. UsMarkVisited inserts $\mathcal{I}$ into the hash table of visited uncertainty states. The algorithm loops (lines 3-11) until a solution is found or all the search space has been exhausted. First, an annotated uncertainty state $\langle Us\ .\ \pi\rangle$ is extracted from the open pool (line 4) by ExtractBest. The uncertainty state is expanded by FwdExpandUsSTable, computing the corresponding UsS table (with length-one sequences). The resulting UsS table is traversed as explained in previous section, accessing with each

```
    procedure SemiSymFwdSearch($\mathcal{I}$,$\mathcal{G}$)
0   begin
1     OpenUsPool := $\{\langle \mathcal{I} . \epsilon \rangle\}$; UsMarkVisited($\mathcal{I}$);
2     Solved := $\mathcal{I} \subseteq \mathcal{G}$; Solution := $\epsilon$;
3     while (OpenUsPool $\neq \emptyset \wedge \neg$Solved) do
4         $\langle Us . \pi \rangle$ := ExtractBest(OpenUsPool);
5         UsSTable := FwdExpandUsSTable(Us);
6         UsSList := PruneListUsSTable(UsSTable);
7         for $\langle Us_i . \alpha_i \rangle$ in UsSList do
8             if $Us_i \subseteq \mathcal{G}$ then
9                 Solved := True; Solution := $\pi; \alpha_i$ ; break;
10            else Insert($\langle Us_i . \pi; \alpha_i \rangle$,OpenUsPool) endif;
11    end while
12    if Solved then return Solution;
13    else return Fail;
14  end
```

Fig. 5. The semi-symbolic, forward search algorithm.

uncertainty state the hash table of the already visited uncertainty states, discarding all the occurrences of present uncertainty states, and marking the new ones. The primitive PruneListUsSTable is a version of PruneUsSTable that returns the explicit list of the UsS pairs in the pruned UsS table. Each of the resulting uncertainty states is compared with the set of target states $\mathcal{G}$. If $Us_i \subseteq \mathcal{G}$, then the associated input sequence $\pi; \alpha_i$ is a solution to the problem, the loop is exited and the sequence is returned. Otherwise, the annotated uncertainty state $\langle Us_i . \alpha_i; \pi \rangle$ is inserted in OpenUsPool and the loop is resumed. If the OpenUsPool becomes empty and a solution has not been found, then a fix point has been reached, i.e. all the reachable space of uncertainty states has been covered, and the algorithm terminates with failure. Depending on ExtractBest and Insert, different search strategies (e.g. depth-first, breadth-first, best-first) can be implemented.

Figure 6 shows the *fully-symbolic, backward* search algorithm. The algorithm relies on sequence variables for a symbolic representation of the input sequences, and recursively expands the UsS tables, thus implementing a breadth-first symbolic search. The algorithm proceeds from $\mathcal{G}$ towards $\mathcal{I}$, exploring a search space built as in figure 3. The array *UsSTables* is used to store the UsS tables representing the levels of the search associated with input sequences of increasing length. The algorithm first checks (line 4) if ϵ is a solution. If not, the while loop is entered. At each iteration, input sequences of increasing length are explored (lines 5 to 8). The step at line 6 expands the UsS table in *UsSTables*$[i-1]$ and stores the resulting UsS table in *UsSTables*$[i]$. UsS pairs which are redundant with respect to the current search are eliminated from *UsSTables*$[i]$ (line 7). The possible solutions contained in *UsSTables*$[i]$ are extracted and stored in *Solutions*

```
    procedure FullySymBwdSearch($\mathcal{I}$,$\mathcal{G}$)
0   begin
1     $i = 0$; UsMarkVisited($\mathcal{G}$);
2     UsSTables[0] := { $\langle \mathcal{G} \ . \ \epsilon \rangle$ };
3     Solutions := BwdExtractSolution(UsSTables[0]);
4     while ((UsSTables[$i$] $\neq \emptyset$) $\wedge$ (Solutions $= \emptyset$)) do
5         $i := i + 1$;
6         UsSTables[$i$] := BwdExpandUsSTable(UsSTables[$i$-1]);
7         UsSTables[$i$] := PruneUsSTable(UsSTables[$i$]);
8         Solutions := BwdExtractSolution(UsSTables[$i$]);
9     done
10    if (UsSTables[$i$] $= \emptyset$) then
11        return Fail;
12        else return Solutions;
13  end
```

Fig. 6. The fully-symbolic, backward search algorithm.

(line 8). The loop terminates if either a solution is found ($Solutions \neq \emptyset$), or the space of input sequences has been completely explored ($UsSTables[i] = \emptyset$).

BwdExtractSolution checks if a UsS table contains a uncertainty state Us_i such that $\mathcal{I} \subseteq Us_i$. It takes in input the BDD representation of a UsS table $UsST_i(\boldsymbol{x},\boldsymbol{\pi}_{[\mathbf{i}]},\ldots,\boldsymbol{\pi}_{[\mathbf{1}]})$, and extracts the assignments to sequence variables such that the corresponding set contains the initial states, by computing $\forall \boldsymbol{x}.(\mathcal{I}(\boldsymbol{x}) \rightarrow UsST_i(\boldsymbol{x},\boldsymbol{\pi}_{[\mathbf{i}]},\ldots,\boldsymbol{\pi}_{[\mathbf{1}]}))$. The result is a BDD in the sequence variables $\boldsymbol{\pi}_{[\mathbf{i}]},\ldots,\boldsymbol{\pi}_{[\mathbf{1}]}$. If the BDD is $False$, then there are no solutions of length i. Otherwise, each of the satisfying assignments of the resulting BDD represents a solution sequence.

The algorithms described here are only two witnesses of a family of possible algorithms. For instance, it is possible to proceed forwards in the fully-symbolic search, and to proceed backwards in the semi-symbolic search.

The algorithms enjoy the following properties. First, they always terminates. This follows from the fact that the set of explored uncertainty sets (stored in the visited hash table) is monotonically increasing: at each step we proceed only if at least one new uncertainty state is generated. The newly constructed UsS table are simplified by removing the uncertainty states which do not deserve further expansion. Since the set of accumulated uncertainty states is contained in Pow($\mathcal{S}$), which is finite, a fix point is eventually reached. Second, a failure is returned if and only if there is no a solution to the given problem, otherwise a solution sequence is returned. This property follows from the facts that in the semi-symbolic search uncertainty states sequences constructed are such that $\langle Us \ . \ \pi \rangle$ enjoy the property $Image[\pi](\mathcal{I}) = Us$. Thus, π is a solution to the problem $\langle \mathcal{I} \ . \ Us \rangle$. In the fully symbolic search uncertainty states sequences constructed are such that $\emptyset \neq Image[\pi](Us) \subseteq \mathcal{G}$. Thus, π is a solution to the problem $\langle Us \ . \ \mathcal{G} \rangle$. The fully symbolic algorithm is also optimal, i.e. it returns plans of minimal length. This property follows from the breadth-first style of the search.

5 Experimental Evaluation

The data structures and the algorithms (semi- and fully-symbolic forward and backward search) have been implemented on top of the symbolic model checker NuSMV [5]. An open hashing mechanism is used to store visited uncertainty states. We present a preliminary experimental evaluation of our approach. We report two sets of experiments. The first ones are from artificial intelligence planning. (The results are labeled with AI in table 1.) FIXi is the generalization of the example system of figure 1 to i devices. For lack of space, we refer to [6] for the description of the other problems. In most cases, the automaton is fully nondeterministic, and there are acceptability conditions for the inputs. The problems are specified by providing the initial and target sets.

The second class of tests is based on the ISCAS89 [7] and MCNC [16] circuit benchmarks, the problem being finding a synchronization sequence, i.e. reaching a condition of certainty (i.e. a single final state) from a completely unspecified initial condition. We ran the same test cases as reported in [12, 13]. In order to tackle these problems, we extended the forward[1] search algorithms (both semi- and fully-symbolic) with an ad-hoc routine for checking if a given uncertainty state is a solution (i.e. if it contains exactly one state).

To the best of our knowledge, no formal verification system able to solve these kind of problems is available, therefore we could not perform a direct experimental comparison. In [6], a detailed comparative evaluation shows that FSB outperforms all the other approaches to conformant planning (based on a decision procedure for QBF [14], on heuristic search [1], and on planning graphs [15]). The results of our approach to searching synchronization sequences appears to be at least as good as the ones in [12, 13]. Normalizing the results with respect to the platform,[2] especially for the problems with longer reset sequences (e.g. planet, sand) we obtain a significant speed up and we are able to return shorter solutions. Furthermore, our approach tackles a more complex problem. Indeed, the approach in [12, 13] is tailored to synchronization problems, and the system is assumed to be deterministic, i.e. uncertainty, intended as the number of indistinguishable states, is guaranteed to be non-increasing. We deal with fully nondeterministic systems, where uncertainty can also grow. Finally, our results were obtained using a monolithic transition relation (although nothing prevents from the use of partitioning techniques).

To summarize, the experimental results seem to confirm the following intuitions. The semi-symbolic approach is often much faster than the fully-symbolic

[1] In order to proceed backwards when searching for a synchronization sequence, the starting point must be the set of all singletons of size $|\mathcal{S}|$. Although possible in theory, the approach seems to be unfeasible in practice.

[2] From the limited information available, we estimate that the results in [12, 13] were obtained on a machine at most 15 times slower than ours.

AI			SSF		FSB	
Name	# FF	# I	L	Time	L	Time
fix2	3	2	5	0.001	5	0.001
fix10	6	4	21	0.001	21	0.440
fix16	6	5	33	0.010	33	56.190
bmtc102l	7	5	18	0.020	18	1.220
bmtc102m	7	5	19	0.020	19	1.190
bmtc102h	7	5	19	0.020	19	1.190
bmtc106l	11	7	18	0.100	14	62.590
bmtc106m	11	7	18	0.100	17	64.970
bmtc106h	11	7	18	0.100	17	64.970
ring2	5	2	6	0.001	5	0.001
ring10	34	2	76	0.100	29	60.940
uring2	5	2	5	0.001	5	0.001
uring10	34	2	29	0.050	29	1.260
cubec	12	3	64	0.040	60	39.210
cubes	12	3	58	0.030	54	16.140
cubee	12	3	42	0.020	42	1.350
omel50	15	3	X	4.400	X	1.380
omel100	17	3	X	67.190	X	8.130

ISCAS'89			SSF_{Sync}		FSS_{Sync}	
Name	#FF	#I	L	Time	L	Time
s1196	18	14	1	3.370	1	0.280
s1238	18	14	1	3.320	1	0.300
s1488	6	8	1	0.010	1	0.000
s1494	6	8	1	0.010	1	0.010
s208.1	8	10	X	0.000	X	0.000
s27	3	4	1	0.010	1	0.000
s298	14	3	2	0.010	2	0.040
s344	15	9	2	0.300	2	6.090
s349	15	9	2	0.300	2	6.100
s382	21	3	1	0.010	1	0.010
s386	6	7	2	0.040	2	0.020
s400	21	3	1	0.010	1	0.000
s420.1	16	18	X	0.120	X	0.000
s444	21	3	1	0.030	1	0.020
s510	6	19		T.O.		T.O.
s526	21	3	2	0.090	2	0.120
s641	19	35	1	1.550	1	0.150
s713	19	35	1	0.540	1	0.150
s820	5	18	1	0.150	1	0.050
s832	5	18	1	0.140	1	0.040
s838.1	32	34	X	0.430	X	0.000

MCNC'91			SSF_{Sync}		FSF_{Sync}	
Name	#FF	#I	L	Time	L	Time
bbara	4	4	2	0.000	2	0.010
bbsse	4	7	2	0.030	2	0.010
bbtas	3	2	3	0.000	3	0.000
beecount	3	3	1	0.000	1	0.000
cse	4	7	1	0.000	1	0.010
dk14	3	3	2	0.000	2	0.000
dk15	2	3	3	0.000	1	0.000
dk16	5	2	4	0.000	4	0.010
dk17	3	2	3	0.000	3	0.010
dk27	3	1	4	0.000	4	0.000
dk512	4	1	5	0.000	4	0.000
donfile	5	2	3	0.000	3	0.000
ex1	5	9	3	0.000	3	0.160
ex2	5	2	X	0.000	X	0.000
ex3	4	2	X	0.000	X	0.000
ex4	4	6	13	0.010	10	1.150
ex5	4	2	X	0.000	X	0.000
ex6	3	5	1	0.000	1	0.000
ex7	4	2	X	0.000	X	0.000
keyb	5	7	2	0.010	2	0.010
lion9	4	2	X	0.000	X	0.000
mark1	4	5	1	0.000	1	0.000
opus	4	5	1	0.000	1	0.000
planet	6	7	20	0.110		M.O.
s1	5	8	3	0.020	3	0.800
s1a	5	8	3	0.020	3	0.810
s8	3	4	4	0.000	4	0.010
sand	5	11	19	0.190		T.O.
tav	2	4	X	0.020	X	0.000
tbk	5	6	1	0.080	1	0.000
train11	4	2	X	0.000	X	0.000

The experiments were executed on an Intel 300MHz Pentium-II, 512Mb RAM, running Linux. #FF and #I are the number of boolean state variables and inputs in the system automaton. SSF and FSB are the semi-symbolic forward and the fully-symbolic backward algorithms. SSF_{Sync} and FSF_{Sync} are the semi-symbolic and fully-symbolic forward search algorithms extended with the ad-hoc termination test for synchronization sequences. Times are reported in seconds. T.O. means time out after 1 hour CPU. M.O. means memory limit of 500Mb exhausted. L is the length of the solution found. X means that the problem admits no solution.

Table 1. Experimental results

one, when a solution exists. This appears to be caused by the additional sequence variables, and by the breadth-first style of the search. However, the fully-symbolic approach appears to be superior in discovering that the problem admits no solution, and returns sequences of minimal length. Forward search (either fully- or semi-symbolic) is usually inferior to backward (with some notable exceptions).

6 Related Work and Conclusions

In this paper we have presented a new approach to the problem of searching powerset automata which tackles the exponential blowup directly related to the powerset construction. Our approach combines techniques from symbolic and explicit-state model checking, and allows for different, complementary search strategies. The work presented in this paper is based on the work in [6], developed in the field of Artificial Intelligence planning, where fully symbolic search is described. In this paper we extend [6] with semi-symbolic search techniques, and we provide a comparative evaluation of the approaches on a larger set of test cases, including synchronization sequences from the ISCAS and MCNC benchmark circuits. Besides [12, 13], discussed in previous section, few other works appear to be related to ours. In SPIN [8], the idea of combining a symbolic representation with explicit-state model checking is also present: an automaton-like structure is used to compactly represent the set of visited states. In [4], an external hash table is combined with a BDD package in order to extract additional information for guided search. In both cases, however, the integration of such techniques is directed to standard model checking problems.

The work presented in this paper will be extended as follows. An extensive experimental evaluation, together with a tighter integration of optimized model checking techniques, is currently being carried on. Then, different search methods (e.g. combining forward and backward search, partitioning of UsS tables) will be investigated. Furthermore, the approach, currently presented for reachability problems, will be generalized to deal with LTL specifications. Finally, the case of partial observability (i.e. when a limited amount of information can be acquired at run time) will be tackled.

References

1. B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, 5^{th} *International Conference on Artificial Intelligence Planning and Scheduling*, pages 52–61. AAAI-Press, April 2000.
2. T. L. Booth. *Sequential Machines and Automata Theory.* J. Wiley, 1967.
3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. G. Cabodi, P. Camurati, and S. Quer. Improving symbolic traversals by means of activity profiles. In *Proceedings of the 31st Conference on Design Automation*, pages 306–311, New York, NY, USA, June 21–25 1999. ACM Pres.

5. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), March 2000.
6. A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research (JAIR)*, 13:305–338, 2000.
7. F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *International Symposium on Circuits and Systems*, May 1989.
8. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
9. Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Company, New York, 1978. ISBN 0-07-035310-7.
10. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.
11. D. Michie. Machine Intelligence at Edinburgh. In *On Machine Intelligence*, pages 143–155. Edinburgh University Press, 1974.
12. C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Conference on Design Automation*, pages 620–623, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.
13. J.-K. Rho, F. Somenzi, and C. Pixley. Minimum length synchronizing sequences of finite state machine. In ACM-SIGDA; IEEE, editor, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 463–468, Dallas, TX, June 1993. ACM Press.
14. J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research*, 10:323–352, 1999.
15. David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 889–896, Menlo Park, July 26–30 1998. AAAI Press.
16. S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, January 1991.