Finding Bad States during Symbolic Supervisor Synthesis

Roberto M. Ziller *

University of Karlsruhe, Department of Computer Science
Institute for Computer Design and Fault Tolerance (Prof. Dr. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
email: ziller@ira.uka.de
http://goethe.ira.uka.de/fmg

Abstract

This paper is about supervisor synthesis, a central issue in solving control problems within the Ramadge-Wonham framework for Discrete Event Systems. As most automata-based methods, this framework is subject to the state explosion problem. The impact of state explosion has been considerably reduced in the area of formal verification through the introduction of symbolic representation techniques, which can also be used in supervisor synthesis. However, the efficiency of a solution is very sensitive to the way the symbolic structures are manipulated at each processing step. This paper explores in detail one such step, namely finding the set of bad states at the start of the synthesis algorithm. Comparing the results for three particular implementations illustrates the importance of a careful choice between different solutions.

Introduction

Supervisor synthesis, introduced by the Ramadge-Wonham framework for Discrete Event Systems [RW87, WR88, ZW90, WW96, BTV97, WW98], can be extremely useful in modern system modelling, especially in combination with formal verification techniques. However, being based on finite automata, the framework is subject to the state explosion problem. Enumerative state space representation often allows solving only problems much smaller than those encountered in real world applications. The successful introduction of symbolic methods to represent the state space in formal verification problems [BCM$^+$90] has motivated other authors to consider them for supervisor synthesis [HWT92, AMP95]. These approaches considered the formal aspects of symbolic synthesis, while the present paper is more implementation-oriented.

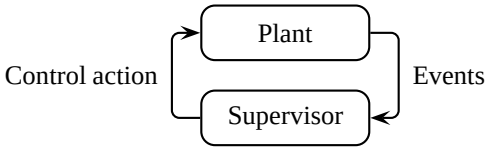
The state explosion problem in the RW-framework was discussed briefly in [Zil01], which illustrates the modelling of different problems using examples that are small enough to be solved by traditional enumerative methods. This paper is a continuation of that work, focusing on the use of symbolic methods to overcome the limits imposed by state explosion. It deals in particular with the isolation of the bad states during synthesis, which has been found to be critical for efficiency. The paper is organized as follows: section 1 is a review of the basic RW–Model; section 2 presents an example that illustrates the synthesis process and also serves to explain the algorithms presented later; section 3 introduces symbolic methods, and section 4 presents the main contribution, consisting of three solutions for finding bad states in a specification and comparing their efficiency. The conclusion comments the results and presents directions for future work.

*Work supported by the Deutsche Forschungsgemeinschaft (German Research Society) within the project Design and Design Methodology of Embedded Systems

1 The Ramadge-Wonham Framework

This section recalls some basic concepts from the RW-framework presented originally in [RW87]; a comprehensive coverage is given in [CL99]. The basic premise is that controlling a system amounts to restrict its physically possible behaviour in such a way that only a subset of desired actions remains executable. In the following, the system to be controlled will be called *plant*, and the desired behaviour under control, a *specification* for that plant. The restrictive control action is applied to the plant by a *supervisor*, an external agent which tracks the plant's state by observing the events it generates and enables or disables (some of) the events that could occur in each state, as illustrated in figure 1.

Figure 1 The interaction between plant and supervisor



Events are classified into two categories. Those which cannot be prevented from occurring, like system failures, and sensor or alarm signals, are called *uncontrollable*, and those which can be disabled by the supervisor, like the start of some process, are called *controllable*. Because of the presence of uncontrollable events, not every specification can be implemented by a supervisor. The latter can not be constructed when a specification allows the plant to reach a state in which uncontrollable events can occur and, at the same time, forbids the occurrence of one or more of these events. Such specifications are called uncontrollable, while implementable specifications are called controllable. Implementing a supervisor for a controllable specification amounts to use the specification to track the plant's actions and to disable, at each state, the events that appear in the plant but do not appear in the specification.

Given an uncontrollable specification, it is always possible to compute its largest controllable subset. This result can be used in place of the original specification, as long as it is still adequate for the intended objective. Since a controllable specification contains all the information needed to build the supervisor for a given plant, this computation is also referred to as *supervisor synthesis*.

Plants and specifications are represented through finite automata, also called *generators*. A generator is a 5-tuple $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$, where Σ is a set of event labels, Q is a set of states, $\delta : \Sigma \times Q \to Q$ is a (possibly partial) transition function, $q_0 \in Q$ is the initial state, and $Q_m \subseteq Q$ is a set of *marker states*. It is tacitly assumed that these are chosen so they mark the completion of some task by the system. The event alphabet Σ is partitioned according to the classification of the events: if Σ_c and Σ_u are the sets of controllable and uncontrollable events, then $\Sigma = \Sigma_c \cup \Sigma_u$ and $\Sigma_c \cap \Sigma_u = \emptyset$; the transition function δ is extended in the natural way to process strings from Σ^*.

It is also useful to define the *active event function* $\Gamma : Q \to 2^{\Sigma}$ as:

$$\Gamma(q) = \{\sigma \in \Sigma : \delta(q, \sigma) \text{ is defined}\}. \tag{1}$$

There are two languages associated with a generator G. The set of all possible event sequences, $L(G)$, is called its *generated language*, while the set of sequences that end up in a marker state, $L_m(G)$, is called its *marked language* and stands for the sequences that represent completed tasks. The generated and marked languages of the specification are subsets of those of the plant. Therefore, it is always possible to derive the specification from the product of the plant and some auxiliary automaton. In what follows, let the state space of the plant be denoted by Q and that of the auxiliary automaton by X. Hence the state space of the specification will be a subset of $Q \times X$. Moreover, there exists a function

$$h : Q \times X \to Q \tag{2}$$

mapping each state $(q, x) \in Q \times X$ to state $q \in Q$ in such a way that the plant will be in state q whenever the specification is in state (q, x). This state correspondence is needed to check whether the specification is controllable. This will be the case if and only if every state in the specification allows the occurence of all uncontrollable events defined in the corresponding state of the plant, that is (with Γ extended to the state space $Q \times X$)

$$(\forall (q, x) \in Q \times X)\, \Gamma(q, x) \cap \Sigma_u \supseteq \Gamma(h(q, x)) \cap \Sigma_u. \tag{3}$$

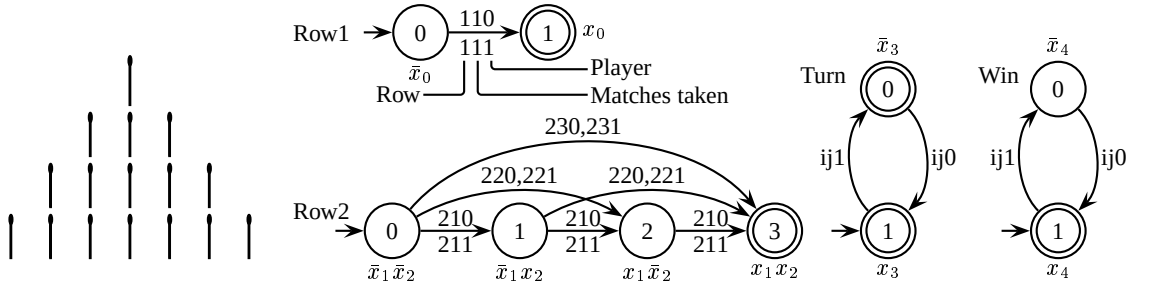
States of the specification that fail to satisfy condition (3) are called *bad states*. The synthesis procedure consists of iteratively removing these states and checking the result for controllability and coaccessibility until a fixpoint is reached [WR87, CL99].

2 Synthesis Example

The following example illustrates the synthesis process. The steps described below are much the same followed in an implementation of a synthesis algorithm using the enumerative approach. The reader may want to contrast these with the symbolic methods used in section 4.

The example shows the synthesis of a winning strategy for the nim game modelled in [Zil01]: given the matches arranged as on the left side of figure 2, each player removes at least one match from one of the rows in each turn. The goal is to force the opponent to take the last match. While the game can be started with any number of rows, only the first two will be used next, in order to keep the automata small enough for drawing. Figure 2 shows the automata used in the model, as well as the binary state variables $x_0 \ldots x_4$ that will be used in sections 3 and 4. Our objective will be to find a winning strategy for player 1 (the player who starts the game). Therefore, the moves of player 1 (odd-numbered events) are assumed to be controllable, while those of player 0 (even) are not.

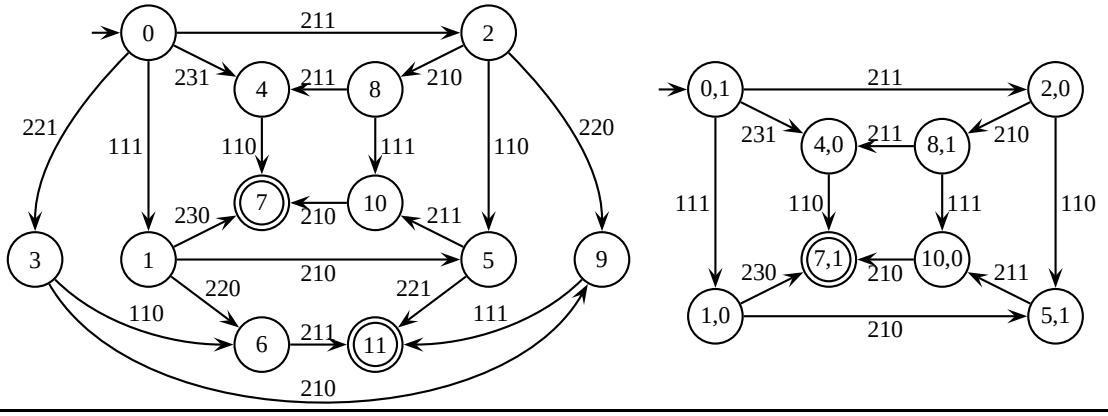
Figure 2 The nim game initial position and the automata for the game with two rows



The synthesis algorithm requires a model of the plant and the specification. The plant is the parallel composition of the automata for the two rows and the auxiliary automaton Turn, which dictates the players must alternate turns. The resulting automaton is shown on the left side of figure 3, where the composite state numbers resulting from the parallel composition have been replaced by a straight sequence. The specification is obtained by computing the product of the plant and the auxiliary automaton Win (which specifies that only player 0 may reach a marker state), and subsequent trimming. The result is shown on the right side of figure 3. The state numbers resulting from the product were maintained, so the first number in each state of the specification identifies the corresponding state of the plant in the sense of expression (2).

The first step of the synthesis process is to check controllability according to condition (3), which reveals states (1,0) and (2,0) to be bad states. Hence the given specification cannot be used to implement a supervisor, but supervisor synthesis can be applied to find its largest controllable subset. In

Figure 3 Plant and specification for the nim game with two rows



the present case, removing states (1,0) and (2,0) does not expose new bad states, so the iteration stops. States (8,1), (10,0), and (5,1) become inaccessible and can be cleaned up. The result is the automaton with states (0,1), (4,0), and (7,1), from which the desired supervisor can be constructed.

The same procedure can also be used to solve larger problems. However, since parallel composition and product appear frequently in the process, the state explosion problem rapidly comes into play. This can be illustrated by adding more rows to the initial configuration of the game. Based on experimental results, the author derived the following expressions for the number of states and transitions of the plant and the specification with n rows. The expressions are valid for $n \geq 2$ unless otherwise indicated.

$$\begin{array}{lll} & \text{Plant} & \text{Specification} \\ \text{\#States} & 2^{n+1}n! - 2^n & 2^{n+1}(n! - 1) \\ \text{\#Transitions} & 2^n(n!n^2 - \frac{n(2n-1)}{2}) & 2^n(n!n^2 - n(2n-1)),\ n \geq 3 \end{array} \tag{4}$$

The state space grows so fast that it is not possible to solve the game for more than five rows using enumerative methods on state-of-the-art personal computers. This problem is the main reason for studying the applicability of symbolic methods to supervisor synthesis. There are several issues which require attention when the states can no longer be handled explicitly, for example which form the correspondence function (2) takes and how to find the bad states before starting iteration. These are the subjects of the following sections.

3 Symbolic Representation of Automata

3.1 Sets and Characteristic Functions

Recall that each subset S of a given set A has an associated *characteristic function* $F_S : A \to \{0, 1\}$, which evaluates to 1 if an element $a \in A$ is contained in S and to 0 otherwise. If the elements of A are encoded using a number of Boolean variables, say, k, the characteristic function becomes a Boolean function $f_S : \{0, 1\}^k \to \{0, 1\}$. This function can be written in disjunctive normal form, as for example in

$$f_S = \bar{x}_0 x_1 \bar{x}_2 + x_0 x_1 \bar{x}_2, \tag{5}$$

which represents the two elements encoded by the combinations $(x_0 = 0, x_1 = 1, x_2 = 0)$ and $(x_0 = 1, x_1 = 1, x_2 = 0)$ from a set with at most eight elements. This is called a *symbolic representation*.

3.2 Binary Decision Diagrams

Binary decision diagrams [Bry86], or BDDs for short, are acyclic graphs that represent binary functions and can therefore represent any finite set through its Boolean characteristic function. The representation is unique for a given ordering of the variables. Moreover, the usual operations on sets can be performed directly on the BDDs representing them, allowing symbolic set manipulation. In many applications, symbolic methods have proven to be much more efficient with respect to memory usage and processing time than their enumerative counterparts.

One of the reasons why BDDs can be very compact is the presence of *don't cares* in the function being represented. A variable x in a binary function f_S is said to be a don't care iff the evaluation of f_S is the same regardless whether x is 0 or 1. For example, variable x_0 is a don't care in expression (5). In what follows, we will need a notation that allows representing don't cares. Drawing the BDDs would be a very low-level and unpractical alternative. Set notation, on the other hand, is too abstract. The natural choice to represent BDDs is therefore the disjunctive normal form of the characteristic function[1], in which don't cares will be represented by a dash. For example, expression (5) becomes

$$f_S = -x_1\bar{x}_2. \tag{6}$$

Many operations on BDDs have been defined and implemented in different programming packages. Besides the traditional set operations, the *existential abstraction* of a variable plays an important role in the sequel. The existential abstraction of variable x turns a binary function f into

$$f' = f\mid_{x=0} + f\mid_{x=1}, \tag{7}$$

thus reintroducing a don't care for the variable x.

3.3 Symbolic Transition Relations

A transition relation can be represented symbolically by a characteristic function in which each transition appears as a minterm. Since a transition consists of an exit state, an event, and an entry state, there will be two sets of variables, called x*-vars* and y*-vars*, to encode the exit states and the entry states, respectively. Another set of variables, called e*-vars*, will be used to encode the events. Figure 2 shows the x-vars used to encode the automata states for the nim game. The events are encoded in ascending order from $\bar{e}_0\bar{e}_1\bar{e}_2$ to $e_0e_1e_2$. The y-vars have the same encoding as the x-vars.

Table 1 shows some transitions from the plant and the specification depicted in figure 3. Note that in the encoding chosen in figure 2, each automaton has its own set of x and y-vars, while the e-vars are common to all. This allows for an efficient representation of automata products. The states of the components are placed side by side in the product, without interfering with each other. This makes the correspondence given by (2) obvious: in order to find the state of a component that corresponds to a given state of the product, it suffices to find, in the component, the state with the same encoding that appears in the product. For example, in the plant state encoded with $x_0\bar{x}_1\bar{x}_2\bar{x}_3$, automaton Row1 is in state 1 (x_0), automaton Row2 is in state 0 ($\bar{x}_1\bar{x}_2$), and automaton Turn is in state 0 ($\bar{x}_3$).

Note also that, because the plant does not use automaton Win, the positions corresponding to the variables x_4 and y_4 appear as don't cares in its transition relation. Strictly speaking, a minterm with $m/2$ don't cares in the x-vars and in the y-vars represents 2^m transitions between $2^{m/2}$ exit states and $2^{m/2}$ entry states. Therefore, the symbolic representation of the plant has more states than those shown on figure 3, and can be seen as a refinement of the original plant containing all the states that can be encoded with the don't cares. This is captured by the notation used in table 1 for the representation of

[1]Note that the disjunctive normal form is an enumerative representation used only to write about BDDs and is never constructed in memory when manipulating them.

Table 1 Some transitions from the plant and specification of figure 3

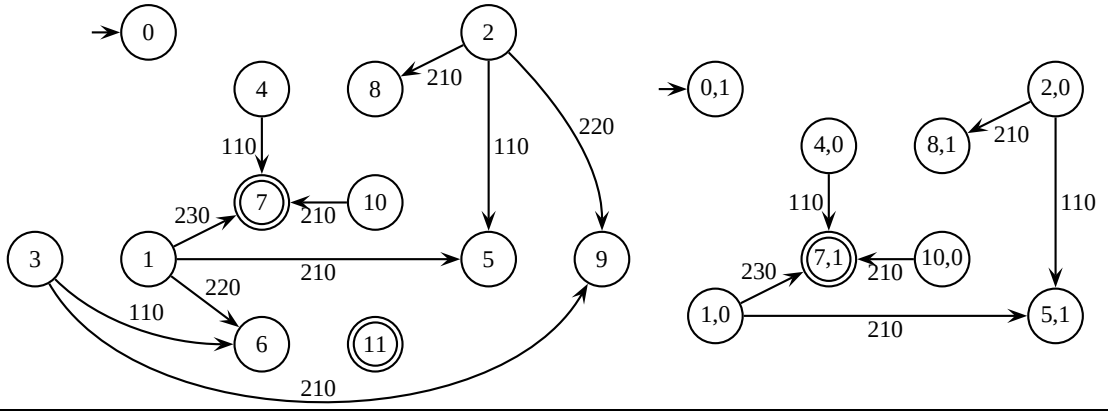
Plant		Specification	
Transitions	Symbolic encoding	Transitions	Symbolic encoding
(0,–),111,(1,–)	$\bar{x}_0\bar{x}_1\bar{x}_2x_3 - \bar{e}_0\bar{e}_1e_2y_0\bar{y}_1\bar{y}_2\bar{y}_3 -$	(0,1),111,(1,0)	$\bar{x}_0\bar{x}_1\bar{x}_2x_3x_4\bar{e}_0\bar{e}_1e_2y_0\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4$
(0,–),211,(2,–)	$\bar{x}_0\bar{x}_1\bar{x}_2x_3 - \bar{e}_0e_1e_2\bar{y}_0\bar{y}_1y_2\bar{y}_3 -$	(0,1),211,(2,0)	$\bar{x}_0\bar{x}_1\bar{x}_2x_3x_4\bar{e}_0e_1e_2\bar{y}_0\bar{y}_1y_2\bar{y}_3\bar{y}_4$
(1,–),210,(5,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - \bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3 -$	(1,0),210,(5,1)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3y_4$
(1,–),220,(6,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3 -$		
(1,–),230,(7,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - e_0e_1\bar{e}_2y_0y_1y_2y_3 -$	(1,0),230,(7,1)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0e_1\bar{e}_2y_0y_1y_2y_3y_4$
(4,–),110,(7,–)	$\bar{x}_0x_1x_2\bar{x}_3 - \bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3 -$	(4,0),110,(7,1)	$\bar{x}_0x_1x_2\bar{x}_3\bar{x}_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3y_4$

the plant's transitions. The additional states can be ignored in operations that involve the plant only, since the BDD would behave as if the unused variables did not exist. However, relating the plant to the specification in section 4 will bring the additional states into play, because the specification contains only a proper subset of them.

4 Finding Bad States

We introduce the problem with a visual approach based on figure 3. According to condition (3), only the uncontrollable transitions shown in figure 4 are needed to decide whether a state is bad. The bad states become apparent after subtracting the transitions of the specification from those of the plant: each bad state will have some outgoing transition in the plant which was not matched in the corresponding state of the specification. Note that state 3 will be erroneously be identified as a bad state. However, since this cannot impact the result of subtracting the bad states from the specification later on, this kind of state can be tolerated among the bad states.

Figure 4 The uncontrollable transitions from plant and specification



The proposed visual approach fails when the specification is refined in such a way that two or more states correspond to the same state of the plant. In contrast, the information contained in the symbolic transition relations will allow us to solve the problem in either case.

The algorithms that follow will start from the transition relations reduced to the uncontrollable events, as shown in figure 4. These can be obtained by intersecting the transition relations with the set of uncontrollable events. In table 1, this has the effect of preserving only the transitions with even-numbered events. The latter will be used to illustrate how the algorithms work.

4.1 Exiting Transitions

Condition (3) is about which events leave a given state. Symbolically, it refers to the exit states and to the events, while it does not matter into which entry state a transition is going. Hence the first step is:

1. Abstract all y-vars from the plant and from the specification.

The uncontrollable transitions from table 1 are thus modified as shown in table 2.

Table 2 The uncontrollable transitions from table 1 after abstraction of the y-vars

Plant		Specification	
Transitions	Symbolic encoding	Transitions	Symbolic encoding
(1,–),210,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - \bar{e}_0e_1\bar{e}_2$ $-----$	(1,0),210,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{e}_0e_1\bar{e}_2$ $-----$
(1,–),220,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - e_0\bar{e}_1\bar{e}_2$ $-----$		
(1,–),230,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3 - e_0e_1\bar{e}_2$ $-----$	(1,0),230,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0e_1\bar{e}_2$ $-----$
(4,–),110,(–,–)	$\bar{x}_0x_1x_2\bar{x}_3 - \bar{e}_0\bar{e}_1\bar{e}_2$ $-----$	(4,0),110,(–,–)	$\bar{x}_0x_1x_2\bar{x}_3\bar{x}_4\bar{e}_0\bar{e}_1\bar{e}_2$ $-----$

It is tempting to abstract also variable x_4 from the specification, because then none of the additional states encoded by the don't cares would appear after subtracting the specification from the plant. Although this would work in the present example, because x_4 appears only in negated form, it cannot be done in the general case. If the specification were refined in such a way that some of its states used x_4 without the negation, the refining information would be lost. We thus proceed with:

2. Subtract the modified specification transitions from those of the plant.

The result is shown in table 3. Note that the second line of table 2 has been split into two rows to show the two exit states represented through the don't care x_4: (1,0), and (1,1). The latter is absent in the specification, as is (4,1). The exit states of the transitions remaining after subtraction either do not appear in the specification, or have a transition that could not be matched by the specification. The latter are the bad states. The nonexistent states can be tolerated in this set, because they cannot impact the result of subtracting the bad states from the specification in the continuation of the synthesis process. The last step is to isolate the bad states (together with the nonexistent states):

3. Abstract the e-vars from the result of step 2.

Table 3 The result of the subtraction for the exiting transitions approach

Plant - Specification	
Transitions	Symbolic encoding
(1,1),210,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4\bar{e}_0e_1\bar{e}_2$ $-----$
(1,0),220,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0\bar{e}_1\bar{e}_2$ $-----$
(1,1),220,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4e_0\bar{e}_1\bar{e}_2$ $-----$
(1,1),230,(–,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4e_0e_1\bar{e}_2$ $-----$
(4,1),110,(–,–)	$\bar{x}_0x_1x_2\bar{x}_3x_4\bar{e}_0\bar{e}_1\bar{e}_2$ $-----$

This algorithm was used in an implementation of the synthesis process. The results for different sizes of the nim game are shown in table 6. Although these are much better than those for enumerative methods, it was noticeable that the abstraction of the y-vars in step 1 required large amounts of memory and processing time. Somewhat surprisingly, the abstraction of the y-vars *together* with the e-vars requires comparatively few resources. The algorithm presented next was designed to take advantage from this fact.

4.2 Delayed Abstraction

We start again with the situation of figure 4 and the uncontrollable transitions from table 1, and delay the abstraction of the y-vars so it can be done simultaneously with the e-vars. The first step is:

1. Subtract the specification transitions from those of the plant.

The left side of table 4 shows the three transitions resulting from the subtraction (1,–),210,(5,–) - (1,0),210,(5,1), as well as those coming from the expansion of the don't cares in the unmatched transition (1,–),220,(6,–). The exiting states so far are either bad or do not appear in the specification, as it was the case in section 4.1. However, the result from the subtraction (4,–),110,(7,–) - (4,0),110,(7,1), shown on the right side of the table, contains the exiting state (4,0), which would be erroneously identified as a bad state if the e-vars and y-vars were all abstracted next. Instead, we continue with:

2. Abstract from the result of step 1 the e-vars together with the y-vars used in the plant.

This turns the term (4,0),110,(7,0) into (4,0),–,(–,0). States like this can be tolerated in the result, because the alternation imposed by the automaton Win assures they do not exist in the specification.

Table 4 Some transitions resulting from the subtraction for the delayed abstraction approach

Plant - Specification			
Transitions	Symbolic encoding	Transitions	Symbolic encoding
(1,0),210,(5,0)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3\bar{y}_4$	(4,0),110,(7,0)	$\bar{x}_0x_1x_2\bar{x}_3\bar{x}_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3\bar{y}_4$
(1,1),210,(5,0)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3\bar{y}_4$	(4,1),110,(7,0)	$\bar{x}_0x_1x_2\bar{x}_3x_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3\bar{y}_4$
(1,1),210,(5,1)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3y_4$	(4,1),110,(7,1)	$\bar{x}_0x_1x_2\bar{x}_3x_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3y_4$
(1,0),220,(6,0)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3\bar{y}_4$		
(1,0),220,(6,1)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3y_4$		
(1,1),220,(6,0)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3\bar{y}_4$		
(1,1),220,(6,1)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3x_4e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3y_4$		

Table 6 shows that exchanging the algorithm from section 4.1 by the above one dramatically improved performance. However, this solution is limited to cases exhibiting alternation, which rises the question whether a general solution with similar efficiency can be found. Another noticeable fact is that the result of the subtraction in step 1 has a large number of nonexistent transitions. While this does not necessarily increase the size of the associated BDD, it leads naturally to ask whether the subtraction can produce a result without these transitions. The next section gives a positive answer to both questions.

4.3 Relaxed and Refined Transitions

We start from the same point as in the preceding sections. The first step is:

1. Abstract from the specification the y-vars that are not used by the plant.

Although abstracting only y-vars has proven inefficient in section 4.1, the above step is expected not to have a serious impact on performance, because the number of variables used to refine the plant is usually small compared to the total of y-vars. As a result, don't cares are introduced in the entry states of the specification at the same point where there are don't cares in the plant's transition relation. Because of the interpretation given to don't cares as representing more than one transition, we call the result the *relaxed specification transitions*. These can be seen on the right half of table 5.

Next we isolate the exit states of the specification (recall that this was found to require comparatively few resources) and use them to refine the plant:

Table 5 Refined plant and relaxed specification transitions

Refined plant transitions		Relaxed specification transitions	
Transitions	Symbolic encoding	Transitions	Symbolic encoding
(1,0),210,(5,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3-$	(1,0),210,(5,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{e}_0e_1\bar{e}_2y_0\bar{y}_1y_2y_3-$
(1,0),220,(6,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0\bar{e}_1\bar{e}_2y_0y_1\bar{y}_2y_3-$		
(1,0),230,(7,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0e_1\bar{e}_2y_0y_1y_2y_3-$	(1,0),230,(7,–)	$x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4e_0e_1\bar{e}_2y_0y_1y_2y_3-$
(4,0),110,(7,–)	$\bar{x}_0x_1x_2\bar{x}_3\bar{x}_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3-$	(4,0),110,(7,–)	$\bar{x}_0x_1x_2\bar{x}_3\bar{x}_4\bar{e}_0\bar{e}_1\bar{e}_2y_0y_1y_2y_3-$

2. Create an auxiliary BDD by abstracting the e-vars and the y-vars from the specification.
3. Intersect the transition relation of the plant with the auxiliary BDD.

As a result, the exit states of the plant take on the refinement of the exit states of the specification. The resulting transitions, called the *refined plant transitions*, can be seen on the left side of table 5. Now, both sides have the same structure, so no spurious states appear when doing the subtraction. Further, states like state 3 of the plant in figure 3, which do not have a counterpart in the specification, are eliminated in step 3. Hence the result of the subtraction contains only transitions which start from a bad state. Finally, these can be isolated, this time without being mixed with nonexistent states:

4. Subtract the relaxed specification transitions from the refined plant transitions.
5. Abstract the remaining y-vars and the e-vars from the result of step 4.

4.4 Experimental Results

Table 6 summarizes the experimental results obtained for different sizes of the nim game. The enumerative results were obtained with the TTCT[2] program. In this case, time was measured roughly and memory usage could not be estimated. The other results are from programs written in C++ by the author, using the CUDD-2.3.1 package[3] and compiled under GNU's gcc–3.0. All experiments were done on a personal computer running RedHat Linux 7.1, equipped with an 800 MHz Pentium III processor and 512 MB of RAM, except for the first case with eight rows, which was done on a similar machine with 2 GB of RAM. Memory usage is given in the peak number of BDD nodes (1 node = 16 bytes), and time is user time, given in seconds. A blank entry means the problem could not be solved within 2 GB of RAM. The ordering of the variables in the BDDs was such that each x-var, starting with x_0, is followed by its corresponding y-var; after them come the e-vars in ascending order[4]. The table also gives the number of plant states according to equations (4).

Table 6 Experimental results

		TTCT	Exiting transitions		Delayed abstraction		Relaxed & refined	
Rows	Plant states	Time	Time	Node peak	Time	Node peak	Time	Node peak
4	752	<1	0.09	22484	0.09	20440	0.09	20440
5	7648	10	0.35	91980	0.37	63364	0.30	65408
6	92096		2.54	568232	1.44	228928	1.65	233016
7	1290112		27.7	4404820	8.38	1089452	8.09	1094562
8	20643584		404	67815832	120	2418052	102	2398634
9	371588608				1164	2606100	1151	2613454
10	7431781376				61726	8850520	61852	8857674

[2]TTCT is developed by Prof. Wonham's group at the Dept. of Electrical Engineering of the University of Toronto.
[3]CUDD is developed by Fabio Somenzi at the University of Colorado at Boulder.
[4]The variable ordering was rearranged in tables 1 to 5 to improve readability.

Conclusion

The paper shows how symbolic methods can be used for finding bad states in supervisor synthesis as an effective measure to deal with the state explosion problem. While the first approach is probably the worst in any case, the other two have a similar performance, and, as far as delayed abstraction is applicable, which of them is the most appropriate depends on the problem being solved. Testing the algorithms against real-world problems may give more information about this issue. The huge performance differences between the first algorithm and the other two alert to the fact that, among many correct implementations, technical details can make a big difference in the final result.

The same might be true for the continuation of the synthesis process, which could not be detailed in the space available here. It involves eliminating the bad states from the specification, finding new bad states thus exposed, and dealing with marker states. The results shown in the last section come from an implementation obtained after testing more than thirty different versions, which nevertheless still allows for improvement.

The size of a BDD is also known to be sensitive to the variable ordering. Although no extensive reordering tests have been done so far, there is indication that the chosen ordering is at least close to the optimum.

References

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, number 999 in LNCS. Springer Verlag, 1995.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model–Checking: 10^{20} States and Beyond. In *Proc. LICS*, 1990.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[BTV97] N. Buhrke, W. Thomas, and J. Vöge. Ein inkrementeller Ansatz zur effizienten Synthese von Controllern aus Spezifikationen mit temporaler Logik. *Proc. Formale Beschreibungstechniken für verteilte Systeme (A. Wolisz et al., eds.)*, pages 99–108, jun 1997.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Boston, U.S.A., 1999. ISBN 0-7923-8609-4.

[HWT92] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proceedings of the American Control Conference*, Chicago, IL, June 1992.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[WR87] W. M. Wonham and P. Ramadge. On the supremal controllable sulanguange of a given language. *SIAM J. Control and Optimization*, 25(3):637–659, may 1987.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of control of discrete event systems*, 1(1):13–30, 1988.

[WW96] K. C. Wong and W. M. Wonham. Hierarchical control of discrete–event systems. *Discrete Event Dynamic Systems*, 6(3):241–273, 1996.

[WW98] K. C. Wong and W. M. Wonham. Modular control and coordination of discrete–event systems. *Discrete Event Dynamic Systems*, 3:241–273, 1998.

[Zil01] R. M. Ziller. System Modelling Using Marker States in the RW–Framework. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, volume 1, pages 121–130. MoPress, 2001.

[ZW90] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervision in discrete–event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, 1990.