System Modelling Using Marker States in the RW-Framework

Roberto M. Ziller[1]

University of Karlsruhe, Institute for Computer
Design and Fault Tolerance (Group Prof. Schmid)
P.O. Box 6980, D–76128 Karlsruhe, Germany
ziller@ira.uka.de

Abstract

The Ramadge-Wonham framework for Discrete Event Systems provides a methodology to derive formal models for a wide class of systems. The many variations of this framework have in common the assumption that synthesis problems are formulated in terms of languages and finite automata. However, little has been said about how these problem formulations can be achieved in the first place. Examples, both instructive to the newcomer and useful as template solutions, are also less frequent in the literature. This paper focuses on the role played by the final states of finite automata in problem formulation and presents examples of their application to system modelling.

Introduction

The Ramadge-Wonham framework for Discrete Event Systems controller synthesis has advanced to an important position among the various approaches suggested in this research field. The basic layout, with one plant and one controller [RW87], has been extended in many ways, so that the framework now encompasses extensions to modular [WR88, WTHM95, WW98], decentralized [RW92], timed [BW93], and hierarchical systems [ZW90, WW96], as well as infinite-behaviour systems [TW94, BTV97].

In all cases, both the system to be controlled and its desired behaviour under supervision are described using languages and finite state automata. The formulation of supervisor synthesis problems assumes that these descriptions are known. While providing the mathematical foundation necessary for the synthesis process, this approach leaves the translation of informal specifications into finite state automata to the user, which is not always a trivial task. In particular, the choice of the final states of the automata, which are called *marker states* in the RW-context, is crucial to the outcome of automata composition operations and to the controller synthesis.

With potential areas of application as varied as logistics, traffic control, and automated manufacturing, to mention only a few, it is clear that a well-developed modelling methodology would be of interest to a wide audience. Handling so many different cases in a single approach is probably not practicable at a detailed level, but at least some classes of abstract behaviour, like blocking avoidance, scheduling, or reacting to external responses in the form of a winning strategy, appear in many different problems. It seems therefore interesting to look at examples of such abstract behaviours in order

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

to learn how to use them to solve other similar problems whenever they can be identified in a system that is being modelled.

The present paper focuses on the use of marker states in system modelling. Section 1 recalls the needed RW-basics, followed by background knowledge about blocking in section 2. Section 3 deals with automata composition, which is essential to combine different automata, each representing a part of a given specification, into a model that stands for the whole system. The main contribution of the paper is given in section 4, which shows application examples to the three abstract behaviour classes mentioned above. The conclusion analyses the results and presents directions for future work.

1 RW Basics

This section recalls the concepts from the RW-framework that are needed in the sequel. A detailed description of the subject is given in [RW87] and related articles.

The system to be controlled, called *plant*, is represented by a finite state machine which makes state transitions according to the occurrence of events. This state machine is referred to as a *generator*. It is a 5-tuple $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$, where Σ is a set of event labels – the *event alphabet* –, Q is a set of states, $\delta : \Sigma \times Q \rightarrow Q$ is a (possibly partial) transition function defined so that $q' = \delta(\sigma, q)$ is the state entered by the system whenever the event σ occurs in state q, q_0 is the initial state, and $Q_m \subseteq Q$ is a set of *marker states*. These are used to mark the termination of certain event sequences, like those representing the completion of a task by the system.

The set of all strings that can be formed with symbols from Σ, plus the empty string ϵ, is denoted by Σ^*, and $\overline{\delta} : \Sigma^* \times Q \rightarrow Q$ is the natural extension of δ, defined so that

$$\overline{\delta}(\epsilon, q) = q \text{ and} \quad (1)$$

$$\overline{\delta}(s\sigma, q) = \delta(\sigma, \overline{\delta}(s, q)), \quad (2)$$

as long as both $q' = \overline{\delta}(s, q)$ and $\delta(\sigma, q')$ are defined. $q' = \overline{\delta}(s, q)$ is the state entered by the system after the events in the string $s \in \Sigma^*$ occur in sequence, starting from state q. The function $\overline{\delta}$ will be written just δ from this point on.

There are two languages associated with a generator. The set of all possible event sequences, called its *generated language*, represents all possible actions the system could possibly execute, and the set of sequences that end up in a marker state, called its *marked language*, stands for the sequences that represent completed tasks.

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 an external control agent, e. g. the start of some process, are called *controllable*. The event alphabet Σ is partitioned accordingly: if Σ_c and Σ_u stand respectively for the sets of controllable and uncontrollable events, then $\Sigma = \Sigma_c \cup \Sigma_u$ and $\Sigma_c \cap \Sigma_u = \emptyset$.

The desired behavior of the system under control action is modelled by a second generator, whose generated and marked languages are subsets of those of the plant. Control action is performed by an external agent called *supervisor*, which observes the events generated by the plant and responds to each of them by updating a set of enabled events, called a *control input* to the plant. Controlling a plant amounts to switch control inputs in response to the generated events in order to restrict its behavior to the desired one.

The framework presents necessary and sufficient conditions for the existence of a supervisor. A specification for a desired behavior that satisfies these conditions is called *controllable*. It is shown that when a specification is not controllable, there is always a least restrictive subset – the *supremal*

controllable sublanguage – of the specification, for which a supervisor exists. This means that if a desired behavior cannot be implemented, one can at least compute its largest feasible subset and use it instead of the original specification, as long as it is still adequate for the intended objective.

2 Blocking and Trimming

This section presents the relevant issues related to marker states and their use in the solution of abstract synthesis problems. The following definitions are needed:

Definition 1 *A state $q \in Q$ in the generator $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$ is said to be* accessible *iff there is a string $s \in \Sigma^*$ so that $\delta(s, q_0) = q$.*

Definition 2 *The* accessible component *of the generator $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$ is the generator $Ac(G) = \langle \Sigma, Q_{ac}, \delta_{ac}, q_0, Q_m \cap Q_{ac} \rangle$, where Q_{ac} is the set of accessible states of G, and δ_{ac} is the transition function δ restricted to the states in Q_{ac}. A generator is said to be* accessible *iff $G = Ac(G)$.*

Definition 3 *A state $q \in Q$ in the generator $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$ is said to be* coaccessible *iff there is a string $s \in \Sigma^*$ so that $\delta(s, q) \in Q_m$.*

A state is coaccessible exactly when there is a path leading from it to a marker state. Notice that taking $s = \epsilon$ easily proves that all marker states are coaccessible.

Definition 4 *The* coaccessible component *of the generator $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$ is the generator $Co(G) = \langle \Sigma, Q_{co}, \delta_{co}, q_0, Q_m \cap Q_{co} \rangle$, where Q_{co} is the set of coaccessible states of G, and δ_{co} is the transition function δ restricted to the states in Q_{co}. A generator is said to be* coaccessible *iff $G = Co(G)$.*

Coaccessibility can be used to deal with blocking. It is reasonable to expect that a well-functioning system will always reach, after a finite sequence of events, a state in which the task it is executing will be finished. If the marker states in the system's description are chosen so they match exactly these finishing states, then the system will be blocking-free exactly when the generator representing it is coaccessible. If it is not, then either there will be a state in which the system gets stuck forever (deadlock) or a group of states through which the system can travel indefinitely without ever reaching a marker state (livelock). System representations may be not coaccessible, while desired behavior specifications, on the other hand, are usually coaccessible.

Definition 5 *The* trim component *of the generator $G = \langle \Sigma, Q, \delta, q_0, Q_m \rangle$ is the generator $Tr(G) = \langle \Sigma, Q_{tr}, \delta_{tr}, q_0, Q_m \cap Q_{tr} \rangle$, where $Q_{tr} = Q_{ac} \cap Q_{co}$ and δ_{tr} is the transition function δ restricted to the states in Q_{tr}. A generator is said to be* trim *iff $G = Tr(G)$ or, equivalently, iff all of its states are both accessible and coaccessible.*

Since generators representing plants and behavior specifications are usually accessible, being trim generally amounts to be coaccessible and therefore the same comments about blocking also apply to this case. From this point on the blocking issue will therefore be related to trim generators.

3 The Synchronous Product

Modelling problems start with an informal description of the system, which has to be translated into an automaton. Experience shows that, except for very small automata (typically below seven states, or sometimes more, if the problem has a high degree of symmetry), the task of manually translating an informal description of a system into an automaton is very error-prone. It is therefore interesting to have a method for constructing large system representations by combining small ones. Only these would be drawn directly from the informal specification. This approach is common to various methodologies, as for example [Hoa85] and [Har87]. The definition below corresponds to what is used throughout the RW-framework.

Definition 6 *The* synchronous product *of the generators* $G_1 = \langle \Sigma_1, Q_1, \delta_1, q_{01}, Q_{m1} \rangle$ *and* $G_2 = \langle \Sigma_2, Q_2, \delta_2, q_{02}, Q_{m2} \rangle$ *is given by the generator*

$$G_1 || G_2 = Ac(\langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \delta_{12}, (q_{01}, q_{02}), Q_{m12} = Q_{m1} \times Q_{m2} \rangle). \tag{3}$$

The transition function δ_{12}*:* $(\Sigma_1 \cup \Sigma_2) \times (Q_1 \times Q_2) \rightarrow Q_1 \times Q_2$ *is defined as:*

$$\delta_{12}(\sigma, (q_1, q_2)) := \begin{cases} (\delta_1(\sigma, q_1), \delta_2(\sigma, q_2)) & \textit{if } \delta_1(\sigma, q_1)! \wedge \delta_2(\sigma, q_2)! \\ (\delta_1(\sigma, q_1), q_2) & \textit{if } \delta_1(\sigma, q_1)! \wedge \sigma \notin \Sigma_2 \\ (q_1, \delta_2(\sigma, q_2)) & \textit{if } \delta_2(\sigma, q_2)! \wedge \sigma \notin \Sigma_1 \\ \textit{undefined} & \textit{otherwise} \end{cases}, \tag{4}$$

where '!' stands for 'is defined'.

Notice that a state (q_1, q_2) will be marked in the product automaton exactly when both q_1 and q_2 are marked in the composed automata, i.e.,

$$(q_1, q_2) \in Q_{m12} \Leftrightarrow q_1 \in Q_{m1} \wedge q_2 \in Q_{m2}. \tag{5}$$

The function δ_{12} can be extended to handle strings from $(\Sigma_1 \cup \Sigma_2)^*$ in the way used for the function δ in equations 1 and 2. It can further be shown that the synchronous product is commutative and associative.

4 Examples

This section presents three classes of problems that can be solved exploring marker states and shows application examples for each of them. In all cases, the objective is to obtain an automaton that represents the final behaviour of the system under consideration.

4.1 Blocking Avoidance

This subsection deals with a situation that can occur when several entities compete for a limited number of shared resources. Suppose each entity is at first in an idle state and that when it receives an order to perform some task, it needs to take several resources out of a resource pool, do some work, and put them back when the task is finished.

The resources are taken one by one, and if the number of free resources in the pool is less than what the entity needs, it is entitled to take all the available resources and to wait until additional ones become free. This configures a potential blocking situation, which would happen whenever the

resource pool gets empty but none of the entities has enough resources to start working and all of them are kept waiting forever.

Blocking can be avoided by properly restricting the access of the entities to the resource pool. A simple way to do so is to forbid the entities to access the resource pool if there are not sufficient resources in there to start working. This approach is always safe, but may be inefficient if accessing a resource is a costly process (e. g. it takes time), because taking all the available resources and waiting does not necessarily conduct to a deadlock. One might therefore want to find a control strategy that is optimal in the sense that access to the resources is inhibited if and only if it leads to blocking.

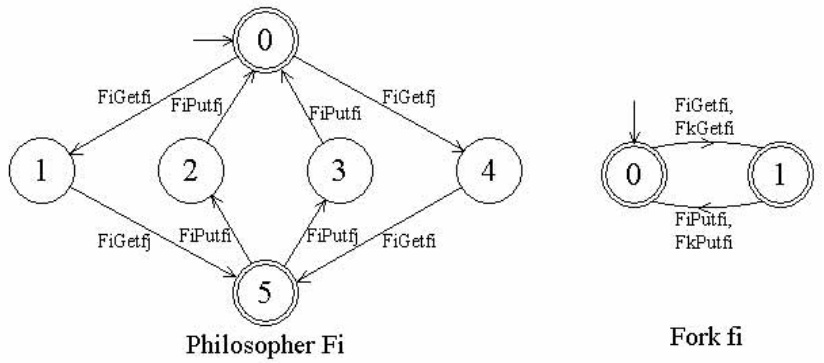
The example below shows how such a potentially blocking system can be modelled by a generator that is not coaccessible, while its trim component can be used to specify the desired behavior.

Dining philosophers problem [Dij71]: N philosophers are sitting at a round table, alternating between thinking and eating. In front of each philosopher is a bowl with food, and between each pair of philosophers is a fork. In order to eat, each philosopher needs to take the forks on his left and on his right, which means that no two adjacent philosophers can eat simultaneously. Once a philosopher takes one fork, he will not put it down before he takes the other one and has eaten for a while. Blocking can occur if, for example, all philosophers take the fork on their left and wait forever for the fork on their right. The goal is to find a strategy that prohibits the philosophers to take a fork if and only if this leads to blocking.

Let's call the philosophers $F_1, F_2, ..., F_N$ and the forks $f_1, f_2, ..., f_N$. Let the forks be laid out so that philosopher F_i has fork f_i on his left and fork f_j on his right, with $j = i \bmod N + 1$. The first step is to draw an automaton to represent each philosopher, as shown on the left side of figure 1. Philosopher F_i may be either thinking (state 0), or holding the fork on his left (1), in which case his next move must be to take the fork on his right, or vice-versa (4), or eating (5, after getting both forks), or holding the right fork after having put the left fork down (2), in which case his next move must be to put the right fork down, or vice-versa (3). The events used to label the transitions are *FiGetfi* (philosopher F_i gets fork f_i), *FiGetfj* (fork f_j), *FiPutfi* (philosopher F_i puts down fork f_i), and *FiPutfj* (fork f_j). It is also assumed that taking a fork is a controllable event, while putting it down is not.

The right side of the figure indicates how the forks are modelled. The events are chosen so that fork f_i can be taken by philosophers F_i and F_k, with $k = (i + N - 2) \bmod N + 1$. These automata dictate that getting a fork and putting it down must follow each other in sequence. Without this, philosophers F_i and F_k could take the same fork without putting it down, a violation of the problem's assumptions.

Figure 1 Automata for the dining philosophers problem



The system to be controlled is represented by

$$System = F_1||...||F_N||f_1||...||f_N. \tag{6}$$

Special attention has to be given to the choice of marker states and to how it affects the coaccessibility of the product automaton. It is clear that state 5 (eating), which represents the goal to be achieved, must be a marker state. But if this were the only such state, then the resulting automaton would have no marker states at all, because it's not possible that all philosophers eat simultaneously and, according to equation 5, a state is marked in the product automaton iff all of the corresponding states in the composing automata are also marked. However, another wanted condition is that all philosophers may eventually go back to their initial (thinking) state, which requires this state to be marked as well. As for the intermediate states 1 to 4, we want a philosopher in one of these states to be able to proceed either to the eating or to the thinking state, but we do not consider them to represent the completion of a task. We therefore leave these states unmarked. This implies that exactly the states in which some philosophers are eating and the others are thinking will be marked in the product automaton. Notice that this does not exclude states in which there are some philosophers eating and others holding a fork. It only means that such states will not be marked. As a consequence, every state in which the system is blocked (any state in which the philosophers are stuck forever waiting for a fork) cannot be marked and is therefore not coaccessible.

As for the choice of marker states in the automata representing the forks, we note that the choice of marker states for the philosophers already gives a correct description of the problem, so what we want from the automata describing the forks is that they do not interfere with what has already been set up. In order to do so, we mark all of their states. Equation 5 implies that the marker states of the automaton resulting from the synchronous product of $F_1||...||F_N$ with $f_1||...||f_N$ are exactly those which correspond to a marker state in $F_1||...||F_N$.

The specification for the desired behavior is given by the trim component of *System* in equation 6. This specification turns out to be controllable, which means that no synthesis procedure is needed. The automaton for the desired behavior already is the solution.

4.2 Scheduling

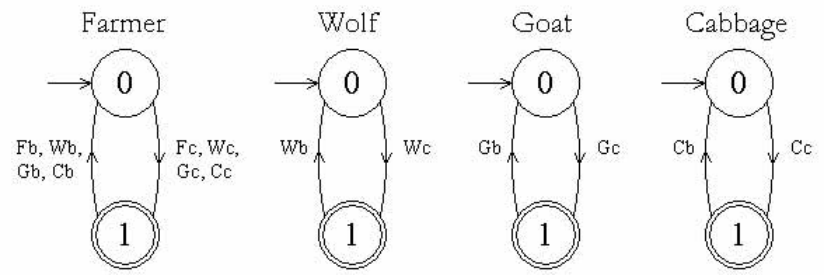
This subsection considers finding a sequence of commands that has to be issued to a system under control in order to take it from a given initial state to a desired final state. It is assumed that the system always moves to a known state after receiving a command and only leaves this state when stimulated by a new command. This means that, as long as one knows how long to wait before issuing the next command, no feedback from the plant in form of uncontrollable events is required. Since commands are modelled as controllable events, all the events in the system specification will be controllable.

Further, one is generally interested in executing any task with as few actions as possible. This means that we choose the shortest solution when faced with more than one possibility. If there are more than one such solutions, the choice is arbitrary. This kind of problem can be solved by creating an automaton which includes the initial state and the desired final state and by finding the shortest paths between them. This can be illustrated by the farmer, wolf, goat, and cabbage problem.

Farmer, wolf, goat, and cabbage problem: a farmer with his wolf, goat, and cabbage comes to a river he wishes to cross. There is a boat, but it only has room for two, and the farmer is the only one that can row. If he leaves the goat and the cabbage on the same shore, the goat will eat the cabbage. Similarly, if the wolf and the goat are together without the farmer, the wolf will eat the goat. The goal is to find a sequence of crossings so the farmer can reach the opposite side safely with all his goods.

We start solving this problem by modelling the farmer, wolf, goat and cabbage by the automata shown in figure 2.

Figure 2 Automata for the farmer, wolf, goat, and cabbage problem



In each automaton, state 0 stands for the near shore, and state 1 for the far shore. The events are named after the initials of the *F*armer, *W*olf, *G*oat and *C*abbage, followed by the letters *c* (cross) or *b* (back). Also, the wolf, goat, and cabbage cannot cross the river on their own. That's why the events for these elements appear on the automaton representing the farmer – if one of the other three crosses the river, he must do so as well. In all automata, state 1 is a marker state, which means that the product automaton representing the system, given by

$$System = Farmer||Wolf||Goat||Cabbage, \tag{7}$$

will have exactly one marker state, in which the farmer and all his goods have completed the crossing. The question is whether this state can be reached without going through a forbidden state in which the goat can be preyed by the wolf or the cabbage, consumed by the goat.

The automaton for the desired behavior can be obtained from the *System* automaton in equation 7 by eliminating the forbidden states, e. g. (0,0,1,1) (goat left with cabbage on the far shore). Since state elimination can destroy accessibility, the next step would be taking the accessible component of the resulting automaton. In order to find a solution, we check if the marker state (1,1,1,1) is preserved in the accessible component, which is indeed the case. When looking for the shortest paths from the initial state to the marker state, we find that there are two equivalent solutions:

- Gc, Fb, Wc, Gb, Cc, Fb, Gc, and
- Gc, Fb, Cc, Gb, Wc, Fb, Gc.

4.3 Winning Strategies

This subsection analyses the case in which the system to be controlled has to be driven towards a desired goal while being allowed to react in every physically possible way. This stands in opposition to the scheduling problem presented in subsection 4.2, where the goal was to find one shortest path to the desired state and the nature of the problem implied all events to be controllable. Here the system

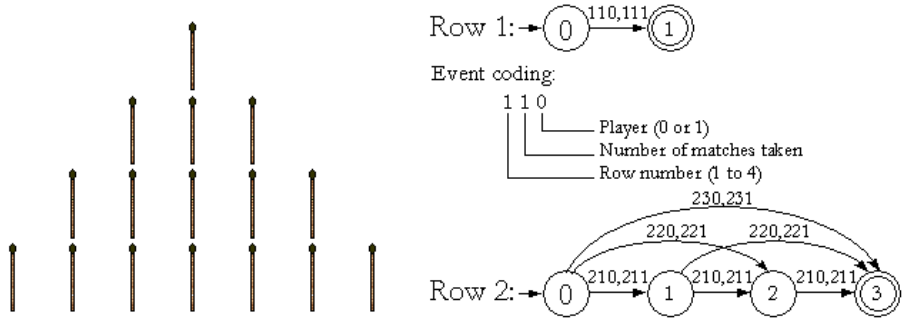
can react in many different ways to each command and the controller's response to each event coming from the system has to guarantee that in all cases the objective can still be achieved. This means that only the commands to the system can be considered controllable, while its reactions must be modelled as uncontrollable events.

The description of the above situation is analogous to that of a two-player zero sum game like tic-tac-toe, checkers or chess. The game is played according to a set of rules that define all legal moves. The system to be controlled is represented by the opponent, and driving it to the desired goal is equivalent to playing according to a winning strategy. Hence, supervisor synthesis amounts to finding such a strategy. The chosen example for this case is the synthesis of a winning strategy to the nim game.

The nim game: given the matches arranged as shown on the left side of figure 3, each player removes as many matches as he wants (but at least one) from one chosen row in each turn. It is not permitted to take matches from two or more different rows in the same turn. The goal is to force the opponent to take the last match.

We start by modelling each row separately by an automaton with $n_i + 1$ states, where n_i is the initial number of matches in row i. The automata for the first two rows are shown on the right side of figure 3, the other two being analogous. The event coding is also given in the figure. In the initial state all matches in each row are present, and taking them makes successive transitions to states where fewer matches are left. In each automaton there is one state with no outgoing transitions that is reached when the row becomes empty.

Figure 3 The nim game initial position and the automata for the first two rows

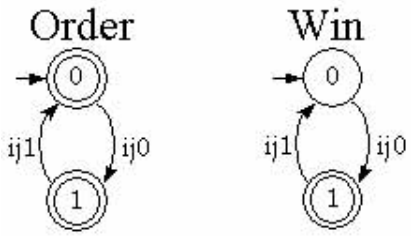


In order to specify that the players alternate turns, we use the automaton *Order*, shown on the left side of figure 4. Here *ij0 (ij1)* stands for all event codings ending with *0 (1)*. The system representation is then given by the automaton

$$System = Row1||Row2||Row3||Row4||Order. \tag{8}$$

The marker states have been chosen so that the final state in each row is marked. This means that the product of the four rows will have a single marker state corresponding to the end of the game. The automaton *Order* has all its states marked so it will not affect the interpretation of the marking established by the rows. The product automaton given by equation 8 has two marker states, one of which can be reached only by player 0 (0 loses), while the other one can only be reached by player 1 (1 loses).

Figure 4 Auxiliary automata for the nim game



Now suppose we are looking for a winning strategy for player 1, that is, we never want this player to reach a marker state. Using the automaton *Win* from figure 4 we form the product

$$OneWins = System||Win, \tag{9}$$

which has the same structure as the automaton *System* with the single difference that the state in which player 1 loses is no longer marked. Since this is a state with no outgoing transitions, it becomes not coaccessible. The specification for the desired behavior is then given by the trim component of *OneWins*, which turns out to be not controllable. Computing the supremal controllable subset results in an automaton that describes a winning strategy for player 1.

For a two-person zero sum game, the existence of a winning strategy for one player implies that the other can have no such strategy. This can be verified by proceeding in an analogous way for player 0. In the final step, the supremal controllable subset turns out to be the empty set, confirming that no winning strategy exists for the player that begins this game.

Conclusion

Problem modelling requires careful choice of the marker states in the automata representing each part of the system. From subsection 4.1 we conclude that marking initial and final states is useful in solving blocking avoidance problems. The example in subsection 4.2 used a single marker state in order to find a path across the automaton, and subsection 4.3 used the synchronous product to affect coaccessibility so that the trim component of the resulting automaton became a description of the desired behavior. In the first two examples, the desired behavior is controllable and therefore already corresponds to the solution. In such cases, the synthesis process merely serves to confirm controllability. In fact, in cases where all events are known to be controllable, the synchronous product can be seen as an independent modelling tool, not necessarily restricted to controller synthesis.

The examples allow for different variations. In section 4.2 the forbidden states were identified manually, and we used information about which states from the composing automata formed each state in the product automaton. One might want to look for another technique that identifies the forbidden states automatically and without that extra information, by bringing in the uncontrollable events “wolf eats goat” and “goat eats cabbage”. The winning strategy approach from section 4.3 can be extended to allow the player with no winning strategy to try to win the game. Of course this would never happen if the opponent makes no mistake, but one can repeat the computations for a winning strategy after every move from the opponent. As soon as a mistake is detected, one would switch into the last built strategy and win the game.

The finite automata approach is known to have serious limitations because of the state explosion problem. The examples presented here are small when compared to real problems, and nevertheless

experimental results show that the number of states and transitions grows quickly (for example 752 states and 5920 transitions for the *System* automaton in expression 8). One one hand, this illustrates the importance of having a composition method like the synchronous product, since it would hardly be possible to assemble the automaton directly. On the other hand, it is a clear indication that real problems require a special computer representation to fit into memory. Symbolic representation, which is already widely used in model checking, can provide a useful answer to this question and is under investigation.

Finally, there is always a risk of using the wrong components when formulating a problem. Therefore, a solution may not correspond to the informal specification, even if it is correct from the point of view of the synthesis process. This suggests that the models described here can be used as entry data for model checking techniques to ensure parts of the informal specification.

References

[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.

[BW93] B. A. Brandin and W. M. Wonham. Modular supervisory control of timed discrete–event systems. *Proceedings of the 32nd IEEE Conference On Decision and Control*, pages 2230–2235, 1993.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Har87] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, jun 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall International, London, U.K., 1985. ISBN 0131532898.

[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.

[RW92] K. G. Rudie and W. M. Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[TW94] J. G. Thistle and W. M. Wonham. Control of infinite behavior of finite automata. *SIAM J. of Control and Optimization*, 32(4):1075–1097, 1994.

[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.

[WTHM95] K. C. Wong, J. G. Thistle, H.–H. Hoang, and R. P. Malhamé. Conflict resolution in modular control with application to feature interaction. *Proceedings of the 34th IEEE Conference On Decision and Control*, pages 416–421, 1995.

[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.

[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.