Generating Controllers
from Discrete-Event Models

Petra Malik

Dept. of Computer Science, University of Kaiserslautern, Germany
dietrich@informatik.uni-kl.de

Supervisors: Prof. Otto Mayer, Dept. of Computer Science, University of Kaiserslautern, Germany; Prof. Wolfram Büttner, Infineon AG, Munich, Germany
Keywords: Discrete-event systems, Supervisory control, Code generation

Abstract. This paper points out problems that occur when controller-code for discrete-event systems is generated from models based on supervisory control. The communication problem and the determinism problem are illustrated by means of a simple example. Some possible solutions for these problems are suggested and compared.

1 Research Area

A *controller* is a device or a piece of software that accepts input from so-called *sensors* and generates output to so-called *actuators* in order to control another system (figure 1). This paper considers the design of logic controllers, i.e. we are interested in the order in which sensors and actuators change their value rather than in the time instants associated with the occurrence of these events. Application domains include manufacturing systems, traffic systems, car electronics, and communication protocols.

The theory of supervisory control initiated by Ramadge and Wonham [RW89] provides an abstract framework for the control of discrete-event systems. The environment (also called plant) is modelled as a generator of a formal language over an alphabet of events. The control feature is represented by the fact that certain events can be disabled by a so-called *supervisor* (figure 2). The general problem of control is to find a supervisor such that the composed system behaviour of the plant and this supervisor meets some specifications. One result of supervisory control theory states that, in the case of regular languages, i.e. finite-state systems, such a supervisor can be effectively computed.

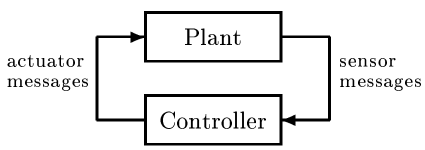

Fig. 1. A controller controlling a plant.

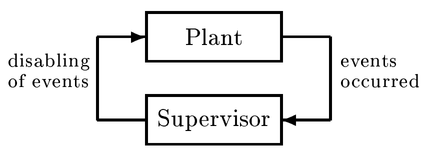

Fig. 2. A supervisor controlling a plant.

Several difficulties [Bal94] have to be mastered when controller-code is generated from supervisory models. In section 2, we illustrate two particular problems, the communication problem and the determinism problem, by means of a simple example. Some possible solutions are suggested and compared in section 3.

2 Problem Description

2.1 An Example

Consider a small example taken from manufacturing. The system consists of two machines A and B which can be started and, after a while, finish their work. We want to design a controller which starts the machines, but ensures that only one machine is working at a time. The controller sends start-messages to and receives finish-messages from the machines. An automaton model for the uncontrolled system is given in figure 3. The initial state `idle` is bordered. Transitions, i.e. arrows, are labelled with event names. At a certain state, an event is accepted (eligible, may be executed) if there exists a transition starting at this state and labelled with the corresponding event name. If this event occurs, the state changes to the destination state of the transition. In the example, the events startA and startB model the starting of the machines. These are *controllable* events, which means that a supervisor may disable them. The events *finishA* and *finishB* model the completion of the work. These are *uncontrollable* events, i.e. they cannot be influenced by a supervisor. Supervisory control theory provides an algorithm for synthesising the least restrictive and nonblocking supervisor which ensures that only one machine is working at a time. The result of this synthesis is shown in figure 4.

How do we obtain a controller from the given model? This can be done by implementing the controlled system behaviour (figure 4). Messages are sent by the controller, when the corresponding controllable event is enabled in the model. Whenever a message is sent or received, the controller updates its state. For instance, at state `idle` the controller sends either message startA or startB and changes its state accordingly. At states `AWorking` and `BWorking`, the controller waits for messages from the plant and updates its state, when a message arrives.

In certain cases, it may be desirable to stop a running machine. In order to achieve this, the model of machine A is extended by the controllable event stopA. A model of the extended machine A is given in figure 5.

Such a situation poses a problem when generating code, since it is not clear how a controller should behave if both controllable and uncontrollable events are possible at the same time. For example, in state `working` of figure 5, the controller can either wait for the uncontrollable event *finishA* to occur, or generate the controllable event stopA.

A natural solution to this problem is to give priority to uncontrollable events, since these events are raised by the plant and cannot be prevented by any controller. In the example, the controller would check whether the plant has sent a *finishA* message, and if so, it would execute the corresponding state transition;

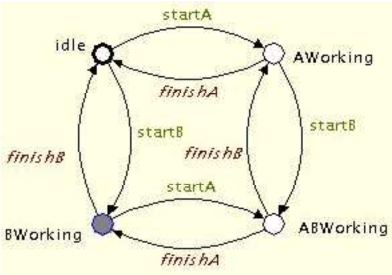

Fig. 3. Uncontrolled behaviour.

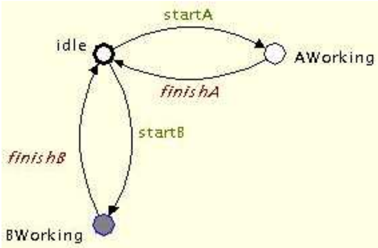

Fig. 4. Controlled behaviour.

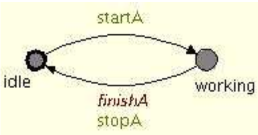

Fig. 5. Machine A with a stopA event.

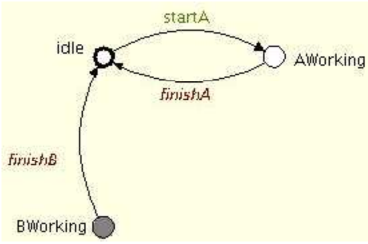

Fig. 6. Restricted controlled behaviour.

otherwise it would send a stopA message. As shown in the next section, this approach causes problems if communication delays are possible.

2.2 The Communication Problem

In many application areas, plant and controller are connected via a network. Delays in the communication channels may cause communication problems. The controller does not get accurate information about the state of the plant, because the corresponding messages do not arrive at the same time the state change occurs. Thus, the controller may decide to send a command based on a wrong assumption about the actual state of the plant.

This is illustrated using the model given in figure 5. Assume that the machine is working and the controller is in state `working`. Since no *finishA* message has arrived, the controller sends a stopA message in order to stop the machine. But before this message is received by the plant, the machine finishes and a *finishA* message is sent to the controller. What happens now? The machine receives a stop message while idle. But this behaviour is not modelled; it is not defined how the machine behaves in this case. The controller receives a *finishA* message while it is in state `idle`. Again, it is not defined how the controller handles this message. Therefore, approaches for the modelling of controllers subject to communication delays are required.

2.3 The Determinism Problem

Many applications require deterministic controllers, i.e. controllers for which each output is uniquely defined by the sequence of events occurred so far. In order to produce a deterministic controller, we have to choose between concurrent controllable events which may be executed at a given state. In the example in figure 4, a choice has to be taken in state `idle` which machine should be started next. A deterministic controller may, for instance, always start machine A restricting the behaviour of the controlled system as shown in figure 6.

The marked states of an automaton are used to describe completed tasks; they represent a set of states which we always want to be reachable by any behaviour—if the system ever enters a state from which no marked state can be reached anymore, it is considered to be blocked. The systems given in figure 3 and 4 are nonblocking, i.e. from every state reachable from the initial state `idle` it is possible to reach the marked state `BWorking`. The system given in figure 6 is blocking, since the marked state cannot be reached. Therefore, by deriving a deterministic controller from a supervisory model, the property to be nonblocking has been lost. We refer to this problem as the *determinism problem.*

3 Results

3.1 Preliminaries

Finite automata are used to describe the behaviour of discrete-event systems and controllers. A (deterministic) automaton is a tuple $A = (\Sigma, Q, \delta, q_0, M)$, where Σ is an alphabet of events, Q is a set of states, $\delta\colon Q \times \Sigma \to Q$ is the (partial) transition function, $q_0 \in Q$ is the initial state, and $M \subseteq Q$ is the set of marked states.

An event is either controllable or uncontrollable, thus the event set Σ is partitioned into the set of controllable events Σ_c and the set of uncontrollable events Σ_u. At a given state $q \in Q$, the eligible event set $\textsc{Elig}_A(q) := \{\, \sigma \in \Sigma \mid \delta(q, \sigma) \text{ is defined} \,\}$ defines the set of accepted events at this state.

3.2 Solving the Communication Problem

One approach to handle the communication problem is to model the communication of plant and controller explicitly. Each message is modelled by means of a sending and a receiving event. The properties of the network are also modelled. This provides the possibility to specify different channel behaviours: whether messages can be interchanged, modified, or removed. However, models with explicit communication tend to be quite large, even for small examples. Furthermore, not all channel behaviours can be modelled easily. For instance, queues (FIFO) with infinite capacity cannot be modelled by means of regular languages.

Balemi [Bal94] proposes a class of supervisors that can be used in a controlled system subject to communication delays. For a special class of plants, Balemi introduces an algorithm solving the controller synthesis problem with communication delays.

In the following, a simple verification approach is suggested in order to identify a class of plants that are robust with respect to the communication problem. This is a special case of Balemi's results, which can be very easily applied.

Observers are used when generating a controller from a supervisory model. As part of a controller, an observer records the state of the plant by observing events occurred so far. The controller only sends messages to the plant on the basis of the state information of the observer. Assuming full synchronisation of plant and controller, the observer can always tell the exact state of the plant. When communication delays may occur, this is not guaranteed anymore. Plant and controller may recognise two messages, one sent by the plant, the other by the controller, in different order. If the future behaviour of the plant depends on the order of these events, the controller cannot make appropriate decisions. The following property, when satisfied by the plant, avoids this problem.

Definition 1. *Let $A = (\Sigma, Q, \delta, q_0, M)$ be a deterministic automaton. The automaton A is Σ_c-Σ_u-commuting, if for all $q \in Q$, $\sigma_c \in \Sigma_c$, and $\sigma_u \in \Sigma_u$ such that $\sigma_c \in \mathrm{E{\scriptsize LIG}}_A(q)$ and $\sigma_u \in \mathrm{E{\scriptsize LIG}}_A(q)$ we have that $\delta(q, \sigma_c\sigma_u)$ and $\delta(q, \sigma_u\sigma_c)$ are both defined and $\delta(q, \sigma_c\sigma_u) = \delta(q, \sigma_u\sigma_c)$.*

A Σ_c-Σ_u-commuting plant accepts a controllable event σ_c and an uncontrollable event σ_u in any order whenever both are accepted at a certain state. Furthermore, the order does not influence the future behaviour of the system. The automaton in figure 5 is not Σ_c-Σ_u-commuting. Although the controllable event stopA and the uncontrollable event *finishA* are accepted in state `working`, the event stopA is not accepted after the occurrence of the event *finishA*.

If a plant is Σ_c-Σ_u-commuting, the following properties are satisfied.

1. The plant accepts any message from a controller that only sends messages on the basis of an observer, even if it is delayed.
2. Each controller, generated from a model containing the plant, accepts any message from the plant, even if it is delayed.
3. If all messages are received, i.e. no messages are pending in the network, the observer can tell the state of the plant.

This enables us to design controllers without considering communications. Nevertheless, controller and plant will accept all messages received, even if communication delays occur.

3.3 Solving the Determinism Problem

In [MCK99] an optimal controller is synthesised for which the highest cost of all paths to a marked state is minimised. This approach solves the determinism problem, but for acyclic discrete-event processes only. In [DMWB02] three properties are given which, when satisfied, ensure that any controller for the system is necessarily nonblocking. In the following, a more general property is introduced. An algorithm is given which checks whether every (deterministic) controller generated from a given model is nonblocking.

We define sets $Q^0, Q^1, \ldots$ of states of an automaton $A = (\Sigma, Q, \delta, q_0, M)$:

$$\begin{aligned}
Q^0 &= M \\
Q^{n+1} &= Q^n \cup \{\, p \in Q \mid \text{there exists } \sigma_u \in \textsc{Elig}_A(p) \cap \Sigma_u : \delta(p, \sigma_u) \in Q^n \,\} \\
&\quad \cup \{\, p \in Q \mid \textsc{Elig}_A(p) \cap \Sigma_c \neq \emptyset \text{ and} \\
&\qquad\qquad \text{for all } \sigma_c \in \textsc{Elig}_A(p) \cap \Sigma_c : \delta(p, \sigma_c) \in Q^n \,\} \\
Q^\infty &= \bigcup_{i \in \mathbb{N}} Q^i
\end{aligned}$$

For each n, the state set Q^n contains all states for which, starting at this state, all deterministic controllers can reach a marked state using strings with length at most n. Thus, *all deterministic controllers for A are nonblocking if and only if all reachable states of A are contained in Q^∞*. In the case of regular languages, i.e. finite automata, there exists an n such that $Q^n = Q^{n+1}$ since $Q^0 \subseteq Q^1 \subseteq Q^2 \subseteq \ldots$ can only increase finitely often. Therefore, Q^∞ is computable in a finite number of iterations. For the automaton in figure 4, we have that $Q^1 = Q^0 = \{\texttt{BWorking}\}$. Since not every reachable state is contained in this set, we conclude that there exists a blocking controller for this model.

3.4 Conclusions and Future Work

We have proposed approaches solving communication and determinism problems occurring when controllers are generated from discrete-event models based on supervisory control. The verification of properties for systems that are robust with respect to the communication problem is not yet completely solved. So far, only the language observed by the controller can be verified, the verification of the language observed by the plant is subject to future work. It is also important to develop efficient algorithms, especially for the fix-point iteration characterising models that are not affected by the determinism problem. Another interesting question is, whether it is possible to synthesise models that are robust with respect to the introduced problems.

References

[Bal94] S. Balemi. Input/output discrete event processes and communication delays. *Discrete Event Dynamic Systems: Theory and Applications*, 4, 1994.

[DMWB02] P. Dietrich, R. Malik, W.M. Wonham, and B.A. Brandin. Implementation considerations in supervisory control. In B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, editors, *Synthesis and control of discrete event systems*, pages 185–201. Kluwer Academic Publishers, 2002.

[MCK99] S. R. Mohanty, V. Chandra, and R. Kumar. A computer implementable algorithm for the synthesis of an optimal controller for acyclic discrete event processes. In *Proc. of 1999 IEEE Int. Conf. on Robotics and Automation*, May 1999.

[RW89] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, January 1989.