Implementation Considerations in Supervisory Control

P. Dietrich[*] R. Malik[†] W.M. Wonham[‡] B.A. Brandin[†]

June 27, 2001

Abstract

With supervisory control theory it is possible to describe controllers which influence the behaviour of a system by disabling controllable events. But sometimes it is desirable to have a controller which not only disables controllable events but also chooses one among the enabled ones. This event can be interpreted as a command given to the plant. This idea is formalized in the concept of an implementation, which is a special supervisor, enabling at most one controllable event at a time. In this paper, some useful properties are introduced, which ensure, when met, that each implementation of a given DES is nonblocking. The approach is applied to a simple batch process example.

1 Introduction

Discrete-event system (DES) theory [RW89, Won99, CL99] provides a framework for describing and analyzing the behaviour of asynchronous controllers and their environment. The environment (also called plant) is modeled as a generator of a formal language over an alphabet of events. An event can either be controllable or uncontrollable. The control feature is represented by the fact that controllable events can be disabled by a so-called supervisor. The general problem of control theory is to find a supervisor such that the closed loop behaviour of environment and supervisor meets some specifications.

When implementing these designs in practice it is sometimes desirable to have a controller which not only disables controllable events, but also chooses exactly one among the set of enabled controllable events which are also physically possible in the plant. This is useful, for instance, in cases where the controllable events chosen by the controller are interpreted as commands for the plant. This setting has been investigated for optimization purposes in [MCK99]. There, an algorithm for synthesizing an optimal controller, i.e. a controller with minimum cost for reaching a marked state from any given state, is introduced.

In our work, arbitrary controllers with unique control action selection are considered in order to get an easily implementable model. Optimizitions of any kind may or may not be used in order to select the desired controller. The problem whith this approach is that a special controller may be blocking even if the given abstract system is nonblocking.

Consider a system with two machines which can be started with controllable events start_1 and start_2 and finish their work with uncontrollable events finish_1 and finish_2. Assume further that we want to ensure that only one machine is working at a time. The controlled system is shown in Figure 1.

In order to implement a controller which actually starts the machines it is not sufficient to disable controllable events. In the case that no machine is working (state a1), a choice must be taken which machine should be started next.

Formally, such a controller can be described as a special supervisor, called an implementation, en-

[*]Dept. of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany, dietrich@informatik.uni-kl.de

[†]Siemens Corporate Research, CT SE 4, 81730 Munich, Germany, {robi.malik|bertil.brandin}@mchp.siemens.de

[‡]System Control Group, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4 Canada, wonham@control.utoronto.ca

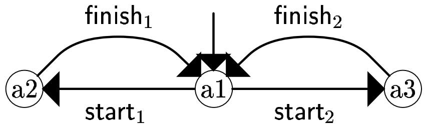

Figure 1: A simple implementation-dependent DES.

abling at most one controllable event at a time. Possible implementations for the DES given in Figure 1 are the supervisor which always disables start_1 in state a1 or the supervisor which disables start_2 at the beginning and after the occurrence of finish_2 and disables start_1 after the occurrence of finish_1.

Assume that state a3 is the only marked state of the system. Then, the given abstract system is nonblocking. Now consider the implementation which always disables start_2. In this case the marked state cannot be reached, and the implementation is blocked. Beside this fact, running only one of the two machines may not be the desired behaviour of the implemented system. It is necessary to refine the model in order to restrict the set of possible implementations.

In this paper, some properties are introduced, which, when checked for the model, can help finding parts of the design which should be refined further. On the other hand, if the properties are met by a DES, then every implementation of the DES is nonblocking. Code for a nonblocking controller can then be generated easily, choosing any implementation.

This paper is organized as follows. Section 2 gives a short introduction to supervisory control theory. The concept of implementation is introduced in Section 3. In Section 4, properties are introduced which ensure that every implementation is nonblocking. The approach is applied to a simple batch process example in Section 5. Conclusions are formulated in Section 6.

2 Supervisory Control Theory

In this section we summarize basic notations of the supervisory control theory introduced by Ramadge and Wonham. For more information see [RW89, Won99, CL99].

2.1 Languages

An *alphabet* is a finite set of symbols. For an alphabet Σ, let Σ^* denote the set of all finite *strings* (or *words*) of the form $\sigma_1\sigma_2\ldots\sigma_k$, where $\sigma_i \in \Sigma$ for $1 \le i \le k$, including the *empty string* ε. A *language* over Σ is any subset $\mathcal{L} \subseteq \Sigma^*$. For $s \in \Sigma^*$, we say that $t \in \Sigma^*$ is a *prefix* of s, and write $t \sqsubseteq s$, if $s = tu$ for some $u \in \Sigma^*$. The *prefix-closure* $\overline{\mathcal{L}}$ of a language $\mathcal{L} \subseteq \Sigma^*$ is the set of all prefixes of strings in $\mathcal{L}$, i.e. $\overline{\mathcal{L}} = \{\, t \in \Sigma^* \mid t \sqsubseteq s \text{ for some } s \in \mathcal{L} \,\}$. The *left quotient* of a language $\mathcal{L} \subseteq \Sigma^*$ by a word $s \in \Sigma^*$ is defined by $\mathcal{L}/s = \{\, t \in \Sigma^* \mid st \in \mathcal{L} \,\}$. The left quotient describes the possible *continuations* of a word in a language.

The *Myhill-Nerode equivalence* of the language $\mathcal{L} \subseteq \Sigma^*$ is an equivalence relation $\mathrm{Nerode}(\mathcal{L}) \subseteq \Sigma^* \times \Sigma^*$ defined as $s \equiv t \bmod \mathrm{Nerode}(\mathcal{L})$ if and only if $\mathcal{L}/s = \mathcal{L}/t$. The Myhill-Nerode equivalence is known to be a right congruence on Σ^*, i.e. for all $s, t, u \in \Sigma^*$ such that $s \equiv t \bmod \mathrm{Nerode}(\mathcal{L})$ we have that $su \equiv tu \bmod \mathrm{Nerode}(\mathcal{L})$.

2.2 Discrete-Event Systems

Discrete-event systems are dynamic systems that evolve in accordance with the abrupt occurrence of physical events. Such systems generally are discrete in time and state space, often asynchronous, and typically nondeterministic.

A discrete-event system (DES) is modeled as a generator of two formal languages over the same alphabet. Formally, it is a tuple $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ where Σ is an alphabet of events, $\mathcal{L}$ is a prefix-closed language over Σ, and $\mathcal{L}_m \subseteq \mathcal{L}$ is another language over Σ, called the *marked language*. The language $\mathcal{L}$ describes all possible behaviours of D, while the marked language of a DES is used to describe completed tasks; it represents a set of words which we always want to be reachable by any behaviour. If the system executes a word in the behaviour which cannot be completed to a string of the marked behaviour, it is considered to be blocked. More formally, a DES $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ is said to be *nonblocking* if $\overline{\mathcal{L}_m} = \mathcal{L}$, otherwise D is said to be *blocking*. In other words, we can say D is nonblocking if for all $s \in \mathcal{L}$ there exists

$t \in \Sigma^*$ such that $st \in \mathcal{L}_m$.

A DES can also be expressed as a *generator*, formally a tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Σ is a finite alphabet of events as above, Q is the *state set* (at most countable), $\delta\colon Q \times \Sigma \to Q$ is the (partial) *transition function*, q_0 is the *initial state*, and $Q_m \subseteq Q$ is the subset of *marker states*. The languages associated with G are $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. The language $\mathcal{L}(G)$ is defined as the set of all strings of events corresponding to sequences of state transitions leading from the initial state to any state of G. The marked language $\mathcal{L}_m(G)$ is the set of all strings of events corresponding to sequences of state transitions leading from the initial state to a marked state. Here, G is another representation for the DES $D = (\Sigma, \mathcal{L}(G), \mathcal{L}_m(G))$. *Transition graphs* are graphical representations of these generators for describing examples used in this paper.

Later we need the concept of the Myhill-Nerode equivalence relation Nerode(D) over a DES $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ which is formally defined as Nerode(D) $=$ Nerode($\mathcal{L}$) $\cap$ Nerode($\mathcal{L}_m$).

2.3 Supervisors

The general problem of control theory consists of finding a supervisor influencing the behaviour of a given system in such a way that it meets the control objectives. A supervisor can only enable or disable controllable events. Uncontrollable events cannot be disabled by a supervisor. Formally, let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Sigma_c \subseteq \Sigma$ be the alphabet of all controllable events of D. A *supervisor* S for D is a function $S\colon \mathcal{L} \to 2^{\Sigma_c}$ from the language of D to the power set of Σ_c. The supervisor maps each word of the language to the set of controllable events which are enabled after the occurrence of that word.

The *controlled system* is denoted by S/D (S controlling D) and its closed loop behaviour is defined as $(\Sigma, S/\mathcal{L}, S/\mathcal{L}_m)$ where $S/\mathcal{L}$ is the smallest language such that

- $\varepsilon \in S/\mathcal{L}$, and
- $s\sigma \in S/\mathcal{L}$ if and only if $s \in S/\mathcal{L}$, $s\sigma \in \mathcal{L}$, and $\sigma \in \Sigma_u \cup S(s)$.

The marked language is defined as $S/\mathcal{L}_m = (S/\mathcal{L}) \cap \mathcal{L}_m$. A supervisor restricts the behaviour of the given DES. Therefore, the language of the controlled system is contained in the language of the uncontrolled system.

Furthermore, we want the controlled system to be nonblocking. A supervisor S (for D) is said to be *nonblocking*, if $\overline{S/\mathcal{L}_m} = S/\mathcal{L}$, i.e. if the closed loop system S/D is nonblocking.

Ramadge and Wonham show that, if there exists a nonblocking supervisor satisfying a given specification, then there exists a least restrictive supervisor with these properties. This least restrictive supervisor S disables as little as possible, i.e. if there is a supervisor S' which meets the requirements and enables the set $S'(s)$ after the occurrence of the word $s \in \mathcal{L}$, then $S'(s) \subseteq S(s)$. Ramadge and Wonham also show that this least restrictive supervisor is computable in the case of regular languages by using a fix-point iteration.

3 Implementations

Assume that a plant model and a (least restrictive) supervisor, which ensures that the specifications are satisfied, are given. In the following, we refer to the given controlled system simply as a DES, also called *abstract model*, and do not distinguish between plant and supervisor model.

If the real system generates uncontrollable and controllable events on its own or is driven by an agent, for instance manually by a human, then the supervisor is easily implementable. The only task of the implemented supervisor, also called *controller*, is to disable controllable events. But it is often the case that the plant does not generate all controllable events on its own without being initiated. Normally, simple machines do not start their work unless the start command (for instance by pushing the start button) is given. In this case it is desirable to have a controller which not only disables controllable events but also initiates the occurrence of particular controllable events. Let Ξ be the set of events which should be initiated by the controller. It makes sense to assume that Ξ is a subset of controllable events.

This does not mean that the event contained in Ξ and chosen by the controller is forced to occur next. It can also happen that an uncontrollable event or a controllable event not contained in Ξ occurs instead. Restrictions on the behaviour of the model are only made with respect to events in Ξ, i.e. those controllable events which are not generated by the plant itself unless being caused by the controller.

We enforce such restrictions on a DES by introducing an *implementation* of the DES. Given a DES D and the subset Ξ of controllable events which should be initiated by the controller. A Ξ-implementation of D can be described as a special supervisor for D which does not restrict the occurrence of events not contained in Ξ, and enables at most one event of Ξ which is also possible in D. Furthermore, it must not disable every event contained in Ξ and possible in D, if there exists one.

Definition 3.1. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let Σ_c be the set of all controllable events of D, and let $\Xi \subseteq \Sigma_c$. A supervisor $I\colon \mathcal{L} \to 2^{\Sigma_c}$ is said to be a* Ξ-implementation *of D if we have for all $s \in \mathcal{L}$:*

(I1) $I(s) \backslash \Xi \subseteq \textsc{Elig}_{\mathcal{L}}(s)$ *and*

(I2) $\textsc{Elig}_{\mathcal{L}}(s) \cap \Xi \neq \emptyset \Rightarrow |I(s) \cap \Xi \cap \textsc{Elig}_{\mathcal{L}}(s)| = 1,$

where $\textsc{Elig}_{\mathcal{L}}(s) := \{\sigma \in \Sigma \mid s\sigma \in \mathcal{L}\}$ *is the set of eligible events after the occurrence of s in $\mathcal{L}$.*

Consider another example with two machines which can be started with controllable events start_1 and start_2 and finish their work with uncontrollable events finish_1 and finish_2. Assume further that the machines can be sent to a self-test with controllable event test and finish the self-test with uncontrollable event done. We want to ensure that the self-test can be performed only if no machine is working and, furthermore, during the self-test no machine should be started. The controlled system is given in Figure 2.

Now we want to implement a controller, which not only disables controllable events but actually starts the machines, i.e. $\Xi = \{\mathsf{start}_1, \mathsf{start}_2\}$. The controllable event test is disabled if necessary but will not be chosen by the controller as a suggestion to occur next, instead it can be generated for instance by human intervention. A possible Ξ-implementation is the supervisor which enables test and start_1 in state a1, no controllable event in state a2, start_2 in state a3, start_1 in state a4, and no controllable event in state a5. A supervisor which disables test in state a1 is not a Ξ-implementation for the DES since test is not contained in Ξ and must not be restricted by a Ξ-implementation. Nor is a supervisor which disables start_1 and start_2 in state a1 a Ξ-implementation since the abstract model allows an event from Ξ to occur, but the considered supervisor does not.

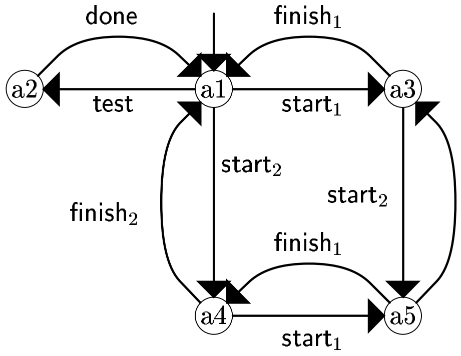

Figure 2: A simple implementation-dependent DES.

In general, the implemented behaviour is a subset of the behaviour of the DES, i.e. the abstract model accepts a larger behaviour than an implementation will do.

4 Ensuring Implementation Independence

In order to implement a given abstract model, we need to refine it, eliminating ambiguity by disabling controllable events. Consider again the introductory example given in Figure 1. In the case that no machine is running we have to specify which machine should be started next. But not all ambiguity has to be eliminated. For instance in the example of Figure 2, we want to start the two machines in any order, but we do not care which one is started first.

The properties discussed in this section help to find possibly critical ambiguity contained in the abstract model. If one of them is violated, a counterexample can be generated, pointing exactly to the problem. Using the counterexample, the model can be refined, and the properties can be checked again. Furthermore, if all properties are met, all possible implementations of this model are nonblocking. Code can be generated easily, choosing any implementation.

4.1 Termination

Most controllers react to an input by only sending a finite sequence of commands. If no input is given for a sufficiently long time, then the system stabilizes: nothing happens until new input is given. This property is formalized in the following definition.

Definition 4.1. *Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$. $\mathcal{L}$ is said to be* Ξ-terminating, *if for all $s \in \mathcal{L}$ there exists an integer n such that we have for each $t \in \Sigma^*$ such that $|t| \geq n$*

$$st \in \mathcal{L} \;\Rightarrow\; t \notin \Xi^*.$$

We think of Ξ as the subset of controllable events for which the choice of the next event has to be taken by the controller. The reason why nonterminating languages (with respect to Ξ) can be a problem when implementing these systems is that an implementation may stay forever in such a loop preventing any progress.

In a Ξ-terminating DES there cannot be an infinite sequence consisting only of events of Ξ. Then, assuming that only events contained in Ξ occur, the system will eventually stabilize, i.e. it will reach a state in which only events not contained in Ξ are possible. The history of a stabilized system is a complete string of events, defined below.

Definition 4.2. *Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$. A string $s \in \mathcal{L}$ is said to be* Ξ-complete *in $\mathcal{L}$, if for all $\sigma \in \Sigma$ it is the case that*

$$s\sigma \in \mathcal{L} \;\Rightarrow\; \sigma \notin \Xi.$$

A Ξ-complete string in $\mathcal{L}$ is a string which cannot be continued with an event contained in Ξ, i.e. only continuations starting with an event not contained in Ξ are possible in $\mathcal{L}$.

The following result shows that, for a Ξ-terminating language, every possible sequence of events continued with events only contained in Ξ will finally result in a Ξ-complete string.

Lemma 4.1. *Let $\mathcal{L}$ be a prefix-closed language over the alphabet Σ, and let $\Xi \subseteq \Sigma$. If $\mathcal{L}$ is Ξ-terminating and $s \in \mathcal{L}$ then the set $C_\Xi(s) = \{\, c \in \Xi^* \mid sc \in \mathcal{L} \,\}$ satisfies the following properties:*

- *$C_\Xi(s)$ is finite and*
- *for each $c \in C_\Xi(s)$ which is maximal in $C_\Xi(s)$ (i.e. for all $c' \in C_\Xi(s)$ we have $c \sqsubseteq c' \Rightarrow c = c'$) it is the case that sc is Ξ-complete in $\mathcal{L}$.*

4.2 Confluence

Assume again, that Ξ is the subset of controllable events for which a choice should be taken by the controller. With the property of confluence we want to ensure that independently of the choices taken, all implementations will reach states, by means of events contained in Ξ only, with the same future. In accordance with the theory of term rewriting [DJ90], we define confluence for a DES as follows:

Definition 4.3. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$. D is said to be* Ξ-confluent, *if for all $s \in \mathcal{L}$, $s_1, s_2 \in \Xi^*$ such that $ss_1, ss_2 \in \mathcal{L}$ there exist $t_1, t_2 \in \Xi^*$ such that $ss_1t_1, ss_2t_2 \in \mathcal{L}$ and $ss_1t_1 \equiv ss_2t_2 \bmod \mathrm{Nerode}(D)$.*

In the introductory example given in Figure 1, assuming $\Xi = \{\mathsf{start}_1, \mathsf{start}_2\}$, the DES is not Ξ-confluent since there exist sequences start_1 and start_2 which, starting from state s1, cannot be continued to states with the same future using sequences contained in Ξ^* only. The DES given in Figure 2 is $\{\mathsf{start}_1, \mathsf{start}_2\}$-confluent, but not $\{\mathsf{start}_1, \mathsf{start}_2, \mathsf{test}\}$-confluent.

If a DES is not only Ξ-confluent but also Ξ-terminating, then, starting at any reachable state of the DES, all Ξ-implementations will reach states with the same future, unless events not contained in Ξ occur. This is formalized in the following lemma.

Lemma 4.2. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$ be a set of controllable events, let I be a Ξ-implementation for D, and let $s \in I/\mathcal{L}$. Furthermore, let D be Ξ-terminating and Ξ-confluent. Then, for each $t \in \Xi^*$ such that $st \in \mathcal{L}$ is Ξ-complete in $\mathcal{L}$, there exists a string $t' \in \Xi^*$ such that $st' \in I/\mathcal{L}$ is Ξ-complete in $\mathcal{L}$, and $st \equiv st'$* mod Nerode(D).

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, let $\Xi \subseteq \Sigma$ be a set of controllable events, let I be a Ξ-implementation for D, and let $s \in I/\mathcal{L}$. Furthermore, let D be Ξ-terminating and Ξ-confluent. Let $t \in \Xi^*$ such that $st \in \mathcal{L}$ is Ξ-complete in $\mathcal{L}$. Let $C = \{c \in \Xi^* \mid sc \in I/\mathcal{L}\}$. Since $I/\mathcal{L} \subseteq \mathcal{L}$ we know that $C \subseteq \{c \in \Xi^* \mid sc \in \mathcal{L}\}$. Furthermore, C is nonempty, since $\varepsilon \in C$, and C is finite by Lemma 4.1. Thus, there exists a string $t' \in C$ which is maximal, i.e. for all $w \in C$ such that $t' \sqsubseteq w$ we have that $t' = w$. Then $st' \in I/\mathcal{L}$ is Ξ-complete in $I/\mathcal{L}$, and by (I2) of Definition 3.1 also Ξ-complete in $\mathcal{L}$. Since D is Ξ-confluent, there exist $u, u' \in \Xi^*$ such that $stu, st'u' \in \mathcal{L}$ and $stu \equiv st'u'$ mod Nerode(D). We know that st and st' are Ξ-complete in $\mathcal{L}$ and therefore $u = u' = \varepsilon$. But this means that $st \equiv st'$ mod Nerode(D). □

4.3 Nonblocking under Control

If a DES is nonblocking then, for all reachable states, there exists a continuation to a marked state. There are no restrictions to this continuation; it can be any string of events. Figure 3 shows a DES which is nonblocking, Ξ-confluent, and Ξ-terminating, where $\Xi = \{c, c'\}$, but still blocking for the Ξ-implementation disabling c at state s1. In order to capture such situations, we now introduce a stronger definition of nonblocking, restricting the continuations to be controlled. A Ξ-*controlled continuation* is one in which events not contained in Ξ occur only if no events of Ξ are enabled. This does not restrict the behaviour of the system, but only strengthens the property of nonblocking. This is formalized in the following definitions.

Definition 4.4. *Let $\mathcal{L}$ be a language over the alphabet Σ, let $\Xi \subseteq \Sigma$, and let $s \in \mathcal{L}$. The string $t \in \Sigma^*$ is called a* Ξ-controlled continuation *of s in $\mathcal{L}$, if*

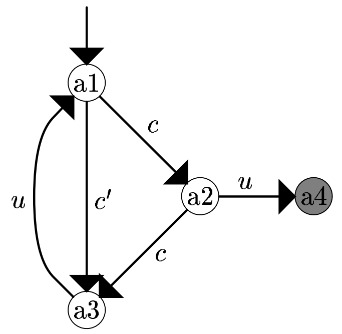

Figure 3: A DES which is not nonblocking under control.

(P1) $st \in \mathcal{L}$ is Ξ-complete in $\mathcal{L}$, and

(P2) for all $a\sigma \sqsubseteq t$ such that $\sigma \notin \Xi$, it is the case that sa is Ξ-complete in $\mathcal{L}$.

This definition can be used to define a strengthened version of nonblocking.

Definition 4.5. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$ be an alphabet. D is said to be* nonblocking under Ξ-control *if for all $s \in \mathcal{L}$ there exists a Ξ-controlled continuation t of s in $\mathcal{L}$ such that $st \in \mathcal{L}_m$; otherwise D is said to be* blocking under Ξ-control.

Because nonblocking under Ξ-control is a specialized kind of nonblocking, the following result is easy to see.

Proposition 4.3. *If a DES is nonblocking under Ξ-control then it is nonblocking.*

Nonblocking under Ξ-control seems to be a rather restrictive property, but the authors think it is a useful property in practice. The reason for this is the following: In the normal definition of nonblocking each continuation leading to a marked state is considered. For instance, starting at the initial state a1 (in Figure 3), the string cu will lead to the marked state a4. But, when interpreting the enablement of events in Ξ as commands given to the plant, events contained in Ξ will occur relatively fast one after the other when enabled. In the given example, and assuming $\Xi = \{c, c'\}$, in order to make the word cu

occur, the uncontrollable event u has to occur just before the controller chooses the output c enabled at state a2; the time interval for this uncontrollable event to occur is very short and depends on the reaction time of the controller. The aim of the above property is to ensure that reaching a marked state must not depend on such time-critical behaviour, but it must always be possible to reach a marked state using continuations, where the controller is not interrupted by the plant (which is normally the more likely behaviour).

Consider the example given in Figure 2 again. Assume the initial state a1 is the only marked state of the system. It is easy to see that this DES is nonblocking. But it is blocking under Ξ-control, where $\Xi = \{\mathsf{start}_1, \mathsf{start}_2\}$, since from state a5 the only Ξ-controlled continuations are $\mathsf{finish}_1\mathsf{start}_1$ and $\mathsf{finish}_2\mathsf{start}_2$ and concatenations of these strings, but all leading to state a5 which is not marked. Such behaviour is problematic because in most execution sequences in practice the controller will restart the machine which has just finished and it is rather unlikely that both machines are ready to restart at nearly the same time (which is the only possibility to reach the marked state). It is not that such sequences cannot happen; probably they can, and we cannot and do not want to prevent them from occurring. But we want to ensure that reaching a marked state is possible without using these sequences.

4.4 Main result

Now we can show that, if a DES is Ξ-terminating, Ξ-confluent, and nonblocking under Ξ-control, then it is nonblocking for every Ξ-implementation. In order to do this we first prove the following lemma.

Lemma 4.4. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$ be a set of controllable events. Furthermore, let D be Ξ-terminating and Ξ-confluent, let I be an implementation of D, and let $s \in I/\mathcal{L}$. Then, for every Ξ-controlled continuation t of s in $\mathcal{L}$ there exists a Ξ-controlled continuation t' of s in $I/\mathcal{L}$ such that $st \equiv st'$ mod Nerode(D).*

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES and let $\Xi \subseteq \Sigma$ be a set of controllable events. Furthermore, let D be Ξ-terminating and Ξ-confluent, let I be an implementation of D, and let $s \in I/\mathcal{L}$. Now let $t \in \Sigma^*$ be a Ξ-controlled continuation of s in $\mathcal{L}$. We show by induction on the number n of events in t, which are not contained in Ξ, that there exists a Ξ-controlled continuation t' of s in $I/\mathcal{L}$ such that $st \equiv st'$ mod Nerode(D).

Base case: $n = 0$ and therefore $t \in \Xi^*$. Since t is a Ξ-controlled continuation we also know that st is Ξ-complete in $\mathcal{L}$. Using Lemma 4.2, we obtain a string $t' \in \Xi^*$ such that $st' \in I/\mathcal{L}$ is Ξ-complete in $\mathcal{L}$ and $st \equiv st'$ mod Nerode(D). Since st' is Ξ-complete in $\mathcal{L}$, $t' \in \Xi^*$, and $st' \in I/\mathcal{L}$ we have that t' is a Ξ-controlled continuation of s in $I/\mathcal{L}$.

Inductive step: $n \to n+1$. Let t contain $n+1$ events of $\Sigma\backslash\Xi$, i.e. $t = a\sigma b$ for some $a \in \Sigma^*$, $\sigma \in \Sigma\backslash\Xi$, and $b \in \Xi^*$. Then a contains n events of $\Sigma\backslash\Xi$, and using the inductive assumption we get a Ξ-controlled continuation a' of s in $I/\mathcal{L}$ such that $sa \equiv sa'$ mod Nerode(D). Therefore, $sa'\sigma b \in \mathcal{L}$ and by definition of I we have that $sa'\sigma \in I/\mathcal{L}$. Furthermore, $sa'\sigma b$ is Ξ-complete in $\mathcal{L}$. Using Lemma 4.2, we obtain a string $b' \in \Xi^*$ such that $sa'\sigma b' \in I/\mathcal{L}$ is Ξ-complete in $\mathcal{L}$ and $sa'\sigma b \equiv sa'\sigma b'$ mod Nerode(D). Now, since Nerode equivalence is a right congruence, $sa\sigma b \equiv sa'\sigma b \equiv sa'\sigma b'$ mod Nerode(D). This equation also implies that $sa'\sigma b'$ is Ξ-complete in $\mathcal{L}$ since $sa\sigma b$ is, and that $a'\sigma b'$ satisfies property $(P2)$ from Definition 4.4, since a' is a Ξ-controlled continuation of s in $I/\mathcal{L}$ and $b' \in \Xi^*$. Since $sa'\sigma b' \in I/\mathcal{L}$ we have that $t' = a'\sigma b'$ is a Ξ-controlled continuation of s in $I/\mathcal{L}$ with the desired property. □

Now, the desired theorem follows easily from the previous lemma.

Theorem 4.5. *Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES, and let $\Xi \subseteq \Sigma$ be a set of controllable events. Furthermore, let D be Ξ-terminating, Ξ-confluent, and nonblocking under Ξ-control. Then, each Ξ-implementation I for D is nonblocking.*

Proof. Let $D = (\Sigma, \mathcal{L}, \mathcal{L}_m)$ be a DES and let $\Xi \subseteq \Sigma$ be a set of controllable events. Furthermore, let D be Ξ-terminating, Ξ-confluent, and nonblocking under Ξ-control, let I be an implementation of D, and let $s \in I/\mathcal{L}$. Since D is nonblocking under Ξ-control there exists a Ξ-controlled continuation t

of s in $\mathcal{L}_m \subseteq \mathcal{L}$. Using Lemma 4.4 we obtain a Ξ-controlled continuation t' of s in $I/\mathcal{L}$ such that $st \equiv st' \bmod \mathrm{Nerode}(D)$. Therefore, since $st \in \mathcal{L}_m$, we also have $st' \in \mathcal{L}_m$. Thus, we obtain $st' \in \mathcal{L}_m \cap I/\mathcal{L} = I/\mathcal{L}_m$. □

This result shows that, in order to obtain nonblocking implementations, it is sufficient to design a model which is terminating, confluent, and nonblocking under control. If these three properties are met, every possible implementation will be nonblocking and can be used to control the system.

So far, it is left to the designer to fix his model if it does not satisfy all three properties. He will be guided by counterexamples, which are automatically computed if one of the properties is not satisfied. These counterexamples point to problems in the design, and usually give hints on how they can be fixed.

Here, it may be desirable to have more support for the designer, by synthesizing a fixed model automatically. Unfortunately, not every system has a most general subsystem which is terminating, confluent, and nonblocking under control. For example, there usually are multiple independent ways of making a non-terminating DES terminating, by deleting different transitions. Therefore, approaches of automatic synthesis will have to deal with multiple optimal solutions which are not comparable to each other.

5 A Small Example

We now discuss a small example taken from [HK94]. We model a dosing unit as it is used in chemical batch plants to supply a defined amount of liquid material to subsequent process units. A dosing unit consists of a tank, an inlet valve A, an outlet valve B, and two sensors, indicating whether the dosing tank is full or empty. In the following, a modular plant model is described.

The two sensors, L1 at the bottom, and L2 at the top of the tank, can either be on or off (see Figure 4). In state a1, when both sensors are off, the tank is empty. It is partially filled in state a2, when sensor L1 is on and sensor L2 is off, and full in state a3, when both sensors are on. The corresponding events

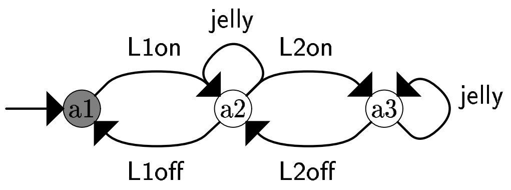

Figure 4: The level measuring sensors.

L1on, L1off, L2on, and L2off, indicating state changes, are uncontrollable. Initially the tank is empty.

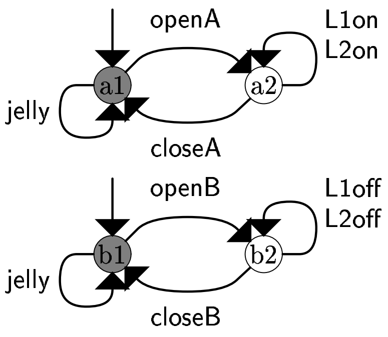

Figure 5: The valves A and B.

The valves A and B can either be open or closed (see Figure 5). The tank can only change its state from empty to partially filled, or from partially filled to full when the inlet valve A is open, and the other way round, it can only change its state from full to partially filled or empty when the outlet valve B is open. The events openA and openB for opening the valves, and closeA and closeB for closing the valves are controllable. Initially both valves are closed.

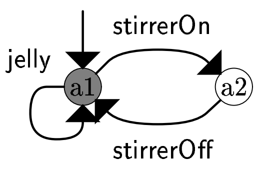

Figure 6: The stirrer.

The fluid must be stirred since it will gelatize, indicated by the uncontrollable event jelly used in Figures 4, 5, and 6, if the substance is not in motion.

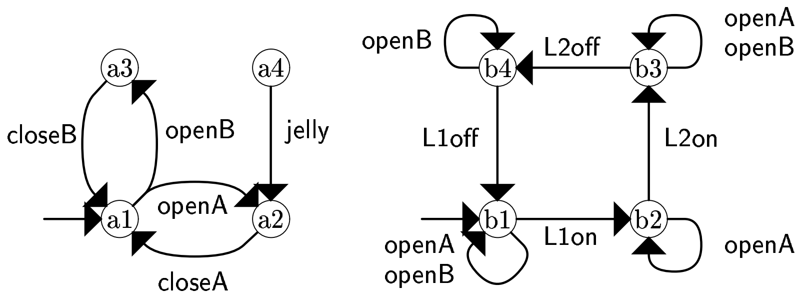

Figure 7: Specifications: a) No material passing through; never jelly. b) Empty-full-empty-cycle.

The stirrer is modeled in Figure 6. It can be switched on, indicated by the controllable event stirrerOn, and switched off, indicated by the controllable event stirrerOff. Initially the stirrer is switched off. The gelatizing process may take place when both valves are closed, the tank is not empty, and the stirrer is not running. Differently from the original example, the gelatizing process can also take place when the tank is full. What happens if the substance turns to jelly is not modeled, since we want to avoid the gelatizing process by the following specifications.

Now we impose certain restrictions to the plant behaviour given so far. The tank must be filled up to the upper level (sensor L2 is on), and then must be discharged until it is empty (sensor L1 is off). It is not necessary to do the filling or discharging in one step, but discharging the tank before the upper level is reached or filling it again before it is emptied, as well as opening both valves at the same time, would not provide the right quantity of substance to the subsequent process and must be avoided. Furthermore, it must never happen that the event jelly occurs. These requirements are specified as automata in Figure 7.

In the original example, the valves are opened and closed manually and are enabled or disabled by a supervisor in order to ensure the specifications. Such a supervisor can be computed, for instance, using the fixpoint iteration given in [WR87]. However, in this paper, we want to design a controller which opens and closes the valves automatically. In order to do this, we select an implementation from the abstract model, which enables at most one controllable event at a time. An enabled controllable event is then interpreted by the plant as a command for the next event. For instance, if openA is enabled by the controller, the valve will be opened next.

But what happens when we compute the least restrictive supervisor and implement the controlled system described above? At the initial state the controllable events openA, openB, and stirrerOn are enabled. The implementation which chooses openB at this state, gets stuck in a loop, for instance, opening and closing valve B forever, preventing any progress. But even if the implementation chooses openA to occur, closeA is enabled immediately after this, and the plant will close valve A as soon as it has opened it. Checking the system for termination will find one of these loops and present it as a counterexample.

There are different ways of avoiding such loops. In this case, some more specifications can be added in order to specify what the controller is supposed to do. We can, for instance, specify that valve A should only be opened when the tank is empty and closed when the tank is full, whereas valve B should only be opened when the tank is full and closed when the tank is empty.

In order to make the example more interesting, we assume that there exist uncontrollable events, perhaps provided from the subsequent process, for starting (event start) and stopping (event stop) the process of filling and emptying the tank. The new plant automaton is given in Figure 8.

The task of the controller is to continue with the filling or emptying of the tank when start happens and stop the process when stop happens. Therefore, we enable the opening of valve A only when start has

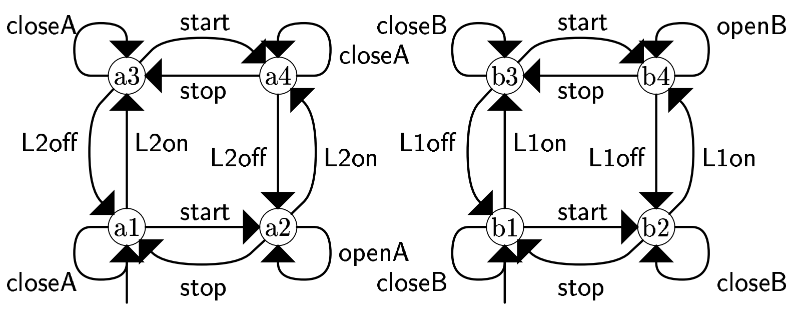

Figure 9: Specifications for the valves

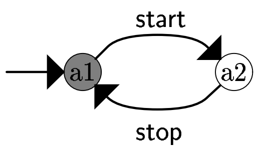

Figure 8: The trigger for starting and stopping the process.

happened and the tank is not yet full. Closing valve A should only occur when the tank is full or the process should be stopped. Valve B should only be opened when the tank is not empty and the process should be continued; otherwise valve B should be closed. This is specified by the automata given in Figure 9.

Checking the abstract system (i.e. the plant system under control of the synthesized supervisor which ensures all the given specifications) for termination will show a loop where the stirrer can be switched on and off and so forth starting, for instance, from the initial state. Assume that we decide to enable the starting of the stirrer only when the system is stopped, and the tank is not empty (since in this case the fluid is not in motion) and only to enable the stopping of the stirrer otherwise. This can be done by adding to the automaton given in Figure 9 the event stirrerOn to the selfloop of state b3, and the event stirrerOff to the selfloops of states b1, b2, and b4.

Now, the controlled system, which we get when synthesizing the least restrictive supervisor, is terminating but not confluent. Whenever the system is started, the tank is full, valve A is open, valve B closed, and the stirrer is started; the controllable events closeA and stirrerOff are possible to occur next. When the stirrer is switched off, valve A cannot be closed any more (it is disabled by the supervisor since otherwise jelly would be possible after closing valve A, but this is prohibited by the specification given in Figure 7). If we close valve A instead, the only controllable events which are possible to occur next are opening the valve B and then switching the stirrer off. Therefore it is not possible to reach states with the same future by means of controllable events only.

What is the problem? We forgot that the fluid must also be stirred when the tank becomes full and should be emptied next. In order to empty it, valve A must be closed first, but then the fluid is not in motion for a short period of time and must be stirred. But we disable the starting of the stirrer in our specifications when the system is not stopped. The counterexample above points to the problem that, when the stirrer is still running and the tank becomes full, one implementation might switch off the stirrer. In this case, the inlet valve A cannot be closed anymore, although the tank is full. Another implementation might close the inlet valve A first and behaves as desired.

Now, we change our specification for the stirrer to the one given in Figure 10. Here, for instance, starting the stirrer is disabled when the tank is empty. Furthermore, starting the stirrer must not occur when the system is switched on, and the tank is not yet full. Now, synthesizing a supervisor for these specifications, will give us a terminating and confluent system, which is also nonblocking under control,

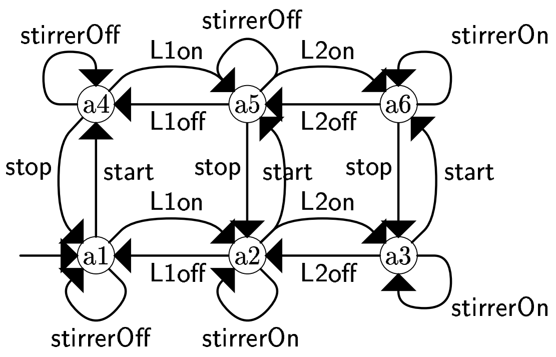

Figure 10: Specifications for the stirrer

under the marking given.

For the sake of demonstration, assume that we have marked states where the tank is empty, the stirrer is switched on, and all valves are closed. The system with these marked states is nonblocking, but not nonblocking under control, since the controllable event stirrerOff is enabled and physically possible at each of these states. Why is this a problem? Actually, the implementation which prefers the event stirrerOff to the events closeA and closeB will never reach one of these states and is therefore blocking.

This example demonstrates how the abstract model can be refined in order to get an implementable model. The introduced properties of termination, confluence, and nonblocking under control are useful checks in order to find ambiguity or nonterminating command sequences in the model.

6 Conclusions

We have provided some properties of DES ensuring that each implementation is nonblocking. They are useful if particular controllable events enabled by the controller are interpreted as commands given to the plant. The implemented controller has to choose among the enabled events in order to produce the next output. The problem is that, even if the given DES is nonblocking, an implementation might not be.

The main theoretical result of this paper is the theorem that, if a DES meets the introduced properties of being terminating, confluent, and nonblocking under control, then all implementations of the DES are nonblocking.

The properties are also useful on their own. If one of them is violated, then there exist ambiguity, or nonterminating command sequences, in the model. Since a counterexample can be generated if such a property is violated, checking these properties can help to find errors in the design.

Further research challenges are the development of efficient algorithms for the introduced properties, especially algorithms which exploit the logical structure often existing in modular designs in order to avoid the state explosion problem. Furthermore, it is conceivable to provide computer guidance for refining the given system when one of the properties is not satisfied. Unfortunately, there does not exist a unique solution for this problem. Further work has to be done in order to find an apropriate subsystem which satiesfies the desired properties.

Other problems which arise in practice when implementing a supervisor, for instance time-delay problems and how to handle the communication between plant and controller, will be discussed in future work.

Acknowledgments

The authors would like to thank Prof. Otto Mayer, Peyman Gohari, and Dr. Franz Regensburger for their support and helpful discussions.

References

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, September 1999.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publisher (North-Holland), 1990.

[HK94] H.-M. Hanisch and S. Kowalewski. Algebraic synthesis and verification of discrete supervisory controllers for forbidden path specifications. In *Proc. of the 4th Intnl. Conf. on Computer Integrated Manufacturing and Automation Technology*, pages 157–162. IEEE Computer Society Press, October 1994.

[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.

[Won99] W.M. Wonham. Notes on control of discrete event systems. Systems Control Group, Dept. of Electrical and Computer Engineering, Univ. of Toronto, Canada; at http://www.control.utoronto.ca/ under "Research", 1999.

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