Locked Discrete Event Systems: How to Model and How to Unlock

Smedinga, Rein

Discrete Event Dynamic Systems: Theory and Applications 2, (1993): 265–297

Locked Discrete Event Systems: How to Model and How to Unlock

REIN SMEDINGA
Department of Computing Science, University of Groningen, Groningen, The Netherlands

Received May 31, 1990; Revised January 25, 1991, November 20, 1991, and June 9, 1992

Abstract. To model qualitative aspects of discrete event systems, i.e., the order of the events is of sole importance, we use a triple consisting of the set of all possible events (the alphabet), the set of all behavior (possible strings of events), and the set of all tasks (completed behavior). We use this view to model synchronous as well as asynchronous connection of systems. Moreover, it is easy to define notions like deadlock and livelock in this view. We give a method to construct a second system that, in connection with the original system, gets rid of its deadlock and/or livelock. A state-space representation is introduced. In this representation computations can be done effectively.

Key Words: discrete event dynamical system, synchronous interaction, asynchronous interaction, trace structure, deadlock, livelock

1. Introduction

Discrete event systems, in which qualitative aspects are of importance, arise in the domains of manufacturing, computer and communication networks, robotics, vehicular traffic, and many others. Modeling such systems can be done at a logical level (consider only the logical order of the events), a temporal level (introducing time), or a stochastic level (introducing probabilities). Here, we consider the logical level.

Logical discrete event systems are first introduced by Hoare and Milner (communicating sequential processes [Hoare 1985] and the calculus for communicating systems [Milner 1980]). Control of such systems is first introduced by Ramadge and Wonham [1987], using a theory based on the theory of automata and languages.

The theory presented here is based on trace theory, introduced in van de Snepscheut [1985]. A logical discrete event system (DES) can then be denoted by two sets: one finite set of symbols representing the events and one set of finite sequences of such symbols, representing the behavior.

The approach as presented here differs from other modeling approaches in the sense that it does not impose any structure on the behavior set, it just is a set of sequences of symbols. It turns out that, even with this simple definition, a DES can be defined, control problems can be formulated, etc. It is not even necessary (for example) to have a prefix-closed behavior.

The first part of this article will introduce DESs using this simple definition. It is a tutorial which captures aspects from other approaches, although stated in a different setting. As an illustration of the usefulness of our approach we give a control problem (how to undo deadlock), formulate it, and give a solution. To be able to perform the necessary computations we also give a state-space form. This enables us to compute the controller effectively. In future articles we would like to study other control problems.

Modeling discrete event systems in a straightforward sense as is done in this article originally comes from Hoare [1985], although he uses recurrent equations, which is only one way of defining the behavior. The ideas of synchronizing discrete event systems using common events that should occur in all systems that are involved at the same moment is also first stated in Hoare [1985] and Milner [1980].

Discrete event systems are also studied by Wonham and Ramadge (for an overview see [1989]). They use a state space to model the behavior of a system. Although the synchronization of systems looks alike there are some (minor) differences: system and controller (supervisor) do not act precisely the same: the controller follows the system and control depends on the state the controller is in. As in this article, Ramadge and Wonham also deal with two aspects of systems: they have $L(G)$, the set of all traces that can be generated by the graph of the system, corresponding to the behavior of the system, and $L_m(G)$, the set of all traces that end in a marker state, corresponding to the task set in our definition.

In this article discrete systems are defined using trace structures. Although trace theory was developed in the context of concurrent programs and found useful in modeling electronic components (see Van de Snepscheut [1985] and Udding [1984]), it seems to be the natural setting to model discrete events in general. In Smedinga [1988] trace theory is first used to model discrete event systems, but without separated task and behavior sets. In Smedinga [1989] the ideas are used to define and solve control problems.

2. Notation

We use a, for pure mathematicians perhaps nonstandard, alternative notation, developed by Dijkstra, that leads to a clear, unambiguous way to display the theory. Also, proofs will be given in a different setting: instead of the (unfortunately too often used) mixture of English and bad mathematical notation we prefer this clearly mathematical style which does not lead to misinterpretations, vague arguments and difficult-to-find errors.

$(\forall x : B(x) : C(x))$ is true if $C(x)$ holds for every x satisfying $B(x)$; e.g., $(\forall x : x \in \mathbb{N}_0 : x \geq 0)$.

$(\exists x : B(x) : C(x))$ is true if there exists an x satisfying $B(x)$ for which $C(x)$ holds; e.g., $(\exists x : x \in \mathbb{N} : x = 10)$.

$\{x : B(x) : y(x)\}$ is the set constructor and denotes the set of all elements $y(x)$ constructed using elements x satisfying $B(x)$; e.g., (with ϵ the empty string), $\{n : n \in \mathbb{N} : a^n b^n\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$.

3. A discrete event system

Logical discrete event systems are systems in which the order in occurrence of events is of importance, not the time of the occurrences. Such a system has a behavior that can be described by giving all sequences of events that are possible. The number of such sequences can, of course, be infinite. We assume that events have no duration, i.e., they occur in infinitesimal time. This implies that no two events can ever occur at the very same time, except if these events *are* the same. In fact, we think of an event as a suddenly change in the state. For example, if a person arrives at some point, we have a suddenly (instantaneous) change in state: before the arrival n persons were present, after the arrival we have $n + 1$ persons. The arrival itself occurs in infinitesimal time: the one moment that person has not yet arrived, the next moment he is present. This assumption is no restriction if we model a system by observing it. The system then is a black box and an observer writes down a symbol each time the corresponding event occurs. The behavior of such a system is now given by all possible sequences of symbols (events) such an observer could write down.

3.1. Trace sets

The essence of a discrete event system lies in its behavior, which is, in fact, a set of sequences of events. Events can be represented by letters, like a, b, or c. Sometimes also words like *arrival*, *stop*, or *enter* are used. A sequence of events can be represented by a sequence of letters (or words) like $a \cdot b \cdot a \cdot c$, meaning that first event a occurs, then event b, then event a again, and, last, event c (see Hoare [1985]). Such a sequence of events is called a trace. A special trace is the empty trace, denoted by ϵ. It represents the nothing-has-happened behavior.

A number of operators on traces can be defined. We mention (with x and y traces, n some natural number, and A some set of events, called the alphabet, see van de Snepscheut [1985]:

concatenation	$x \cdot y$	first x, then y
choice	$x \mid y$	x or y (but not both)
finite repetition	x^n	n times x: $\underbrace{x \,.....\, x}_{n \text{ times}}$
repetition	x^*	zero or more times x
nonempty repetition	x^+	one or more times x
weaving	$x \,,\, y$	shuffling of x and y (see Example 4.2)
restriction	$x \lceil A$	projection on alphabet A

Weaving introduces concurrency. If x and y have no events in common the trace x, y represents the concurrent behavior of x and y. We will return to this subject later on (see Example 4.3).

If letters are used to denote events, we omit the dots for concatenation and write *abac* instead of $a \cdot b \cdot a \cdot c$.

The alphabet restriction is defined by

$$\epsilon \lceil A = \epsilon,$$

$$xa \lceil A = \begin{cases} x \lceil A & \text{if } a \notin A, \\ (x \lceil A)a & \text{if } a \in A. \end{cases}$$

Moreover, we have the operator **pref** that denotes the set of all prefices of a trace:

$$\mathbf{pref}(x) = \{y, z : yz = x : y\}.$$

pref, $\lceil$, and * can also be performed on trace sets. If T is a trace set then

$$\mathbf{pref}(T) = \{y, z : yz \in T : y\},$$

$$T \lceil A = \{x : x \in T : x \lceil A\}.$$

The alphabet generating set A^*, at last, is defined by

$$\epsilon \in A^*,$$

$$x \in A^* \wedge a \in A \Rightarrow xa \in A^*.$$

Sometimes we need the ordering of traces, the length of a trace, or the number of occurrences of some event in a trace. For two traces x and y the order of traces is defined by

$$x \leq y = x \in \mathbf{pref}(y),$$

$$x < y = x \leq y \wedge x \neq y.$$

For some trace x and event a the length of a trace is defined by

$$|\epsilon| = 0,$$

$$|xa| = |x| + 1.$$

Last, for some trace x and events a and b the number of occurrences of a in x is denoted by $x\mathbf{N}a$ and defined by

$$\epsilon\mathbf{N}a = 0,$$

$$(xb)\mathbf{N}a = \begin{cases} x\mathbf{N}a + 1 & \text{if } b = a, \\ x\mathbf{N}a & \text{if } b \neq a. \end{cases}$$

EXAMPLE 3.1. To illustrate the above definitions:

$$(ab\,|\,c^*d) = \{ab, d, cd, ccd, cccd, \ldots\}$$

$$ab^2cb \lceil \{a, c\} = ac,$$

$$\mathbf{pref}(ab) = \{\epsilon, a, ab\},$$

$$\{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\},$$

$$ab^2cb\mathrm{N}b = 3,$$

$$|ab^2cb| = 5,$$

$$ab < ab^2cv.$$

3.2. *Discrete event system*

The simplest way to model a discrete event system is by giving all possible events and all (finite) sequences of events an observer could possibly see when observing the system. So

$$P = \langle \mathbf{a}P, \mathbf{b}P\rangle$$

is such a system, where **a***P* is the *alphabet*, the set of all possible events (and we suppose that we have only a finite number of different events) and **b***P* is its *behavior*, a subset of the set of all finite sequences of events over the alphabet **a***P*, so $\mathbf{b}P \subseteq (\mathbf{a}P)^*$. Moreover, to get a system with a realistic interpretation, we add the restriction that if some sequence is in **b***P*, then also all prefices of that sequence should be in **b***P*. Formally, **b***P* is *prefix-closed*: $\mathbf{pref}(\mathbf{b}P) = \mathbf{b}P$. This is the only restriction on **b***P*. In the theory that follows even this restriction is not needed at all. No other restrictions are imposed on **b***P*, especially, **b***P* need not be regular.

If $\mathbf{b}P = \emptyset$, we say the system is *empty* and if $\mathbf{b}P = (\mathbf{a}P)^*$ we say the system is *complete*. Notice that an empty system has no behavior at all, not even the empty behavior ϵ. If $\mathbf{b}P = \{\epsilon\}$ the system has precisely one behavior: doing nothing.

EXAMPLE 3.2. in general, we will simplify the system when we are modeling it in this way: Consider a system with a farmer, a wolf, a goat, and a cabbage in which the farmer should bring wolf, goat, and cabbage from one side of a river to the other. In order that the wolf does not eat the goat and the goat does not eat the cabbage, the farmer should be aware not to leave wolf and goat or goat and cabbage alone at either side of the river. In fact, this is some old puzzle: how should the farmer solve the problem when he is able to bring at most one item at the time to either side of the river.

Events we can think of are "farmer takes wolf from one side of the river to the other," "farmer takes goat from one side of the river to the other," "farmer takes cabbage from one side of the river to the other," and "farmer goes alone from one side of the river to the other." In fact, each such an event is a simplificatiion of a great number of actions: "pick up the wolf," "put it into the boat," "go into the boat itself," "unrope," "row to the other side," etc. In modeling such a system we consider all these actions to be just one event (and therefore occurring at the same time, changing the state from "items are at some place" to "items are at the same place, except the wolf is now on the other side").

Modeling discrete event systems using $\langle \mathbf{a}P, \mathbf{b}P \rangle$ has one disadvantage: sometimes (mostly when considering more discrete event systems in cooperation with each other) we want to be able to tell when a system has ended legally or when it has ended illegally. A legal ending means that the system has performed a completed task. The above system of farmer, wolf, goat, and cabbage has ended legally if the farmer has succeeded in bringing wolf, goat, and cabbage to the other side of the river. Ending legally (performing a completed task) does not mean the system cannot continue: of course, after a legal behavior more events may occur: the farmer may bring wolf, goat, and cabbage to this side of the river and afterwards to the other side again. After these actions he has, again, performed a completed task (the puzzle does not say anything about efficiency). A system can also end illegally. In that case no task is completed. If also no next event is possible the system has deadlocked.

To distinguish between legal and illegal ending we should add in the definition of a system all completed tasks, i.e., we define a discrete event system as a triple:

$$P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$$

with $\mathbf{a}P$ and $\mathbf{b}P$ as before and $\mathbf{t}P$ the task set of P. Again, to get a realistic interpretation we add the restriction $\mathbf{t}P \subseteq \mathbf{b}P$ (i.e., all tasks of P should belong to the behavior of P). $\mathbf{t}P$ need not be prefix closed (as a completed task need in general not be complete if the last event of it has not yet occurred). Moreover, $\mathbf{b}P$ may be larger than the prefix closure of $\mathbf{t}P$ (i.e., in $\mathbf{b}P$ sequences of events may exist that are no prefix of a sequence of $\mathbf{t}P$). These extra sequences are called the *locked* sequences of the system, because once the system is in such a sequence no task can be completed.

Summarizing, we have

DEFINITION 3.1. A discrete event system P is defined by $P = \langle \mathbf{a}P, \mathbf{b}P, \mathbf{t}P \rangle$ with

$\mathbf{a}P$	alphabet	finite set of events
$\mathbf{b}P$	behavior	$\mathbf{b}P \subseteq (\mathbf{a}P)^* \wedge \mathbf{b}P = \mathbf{pref}(\mathbf{b}P)$
$\mathbf{t}P$	task set	$\mathbf{t}P \subseteq \mathbf{b}P$

The restrictions $\mathbf{b}P = \mathbf{pref}(\mathbf{b}P)$ and $\mathbf{t}P \subseteq \mathbf{b}P$ are added to get a system with a realistic interpretation. Sometimes we will use systems in which these restrictions are not met. Such a system will be called a generalized discrete event system (GDES). It may have, for example, a behavior that is not prefix closed or a task that is no behavior. Such systems go beyond our scope of a discrete event system. Nevertheless, they play a crucial role in some parts of the theory. In Smedinga [1992b] GDESs are used to find a controller system such that the connection with a given system has a behavior that remains between certain limits.

All properties of DESs that are given below do not use the restrictions on behavior and task set, unless stated explicitly. Moreover, all operators that are introduced below, result in a DES if performed on DESs, unless stated otherwise.

In the approach of Ramadge and Wonham a DES is modeled using a deterministic automaton. The resulting languages generated by the automaton correspond to the behavior and task set in our approach. However, our approach does not need a representation of the behavior and task set. It works on the sets itself. Instead of an automaton also other representations can be used to model the system. We mention the well-known command structure from trace theory (see Kaldewaij [1988] and van de Snepscheut [1985]) and the recurrent expressions from Hoare [1985]. The theory derived below does not assume any representation of behavior and task sets.

If $\mathbf{b}P = \mathbf{pref}(\mathbf{t}P)$ we say the system is *lock-free*. Lock-free systems will be denoted by a tuple $\langle \mathbf{a}P, \mathbf{t}P \rangle$, where $\mathbf{b}P$ is left out because it can easily be determined from $\mathbf{t}P$. The term *lock-free* corresponds to the term *nonblocking* as is used in supervisory theory (see Ramadge and Wondam [1989]).

EXAMPLE 3.3. A one-place buffer can be modeled as the following lock-free system:

$$buffer = \langle \{in, out\}, (in \cdot out)^* \rangle.$$

4. Connection of discrete event systems

Now we know how to model one discrete event system we could investigate the cooperation of more discrete event systems. In fact, at the beginning of this article, we have already mentioned the important assumption that no two events could ever occur at the very same time, unless those events are the same (an observer cannot write down two symbols at the same time). From this assumption we might conclude that different discrete event systems should cooperate via common events, events that are the same in these systems (Hoare [1985]; van de Snepscheut [1985]).

But how could one event be the same in, say, two different discrete event systems? Consider for a moment a vending machine that gives coffee after inserting a coin. The machine can be modeled with task set equal to

$$(coin \cdot coffee)^*.$$

The event *coin* represents the insertion of the coin by a person as well as the acceptance of the coin by the vending machine. Also, the event *coffee* represents producing the coffee by the machine as well as accepting the coffee by the person. Both events are representations of a number of actions, partly performed by the vending machine, partly by the person who wants coffee. Those events are common to both vending machine and person and therefore only occur if both systems can engage in it: the vending machine must be able to accept a coin and the person must be able to insert it in order for event *coin* to occur. In this example no coffee can be produced by the vending machine if the person wants tea instead of coffee.

The event *coin* in the discrete event system "vending machine" only occurs if the event *coin* in the discrete event system "person" occurs also (and at the same time). This kind of interaction is called *synchronous interaction* (Hoare [1985]; van de Snepscheut [1985]): a common event occurs only if all systems involved can engage in it.

4.1. Synchronous connection

There is another way of looking at (synchronous) interaction of two systems. We also can say that the resulting behavior of these two systems should be such that, if we restrict this behavior to the alphabet of one of the systems, we should get a behavior from that system. This way of explaining interaction is equivalent to the previous one, but it is simpler to make formal. Therefore, the definition of interaction of discrete event systems makes use of this view:

DEFINITION 4.1. The system that results if two discrete event systems P and R are interacted is called the *connection* of P and R, denoted by $P\,||\,R$, and defined by

$$P\,||\,R = \langle \mathbf{a}P \cup \mathbf{a}R,\ \{x : x \in (\mathbf{a}P \cup \mathbf{a}R)^* \wedge x\lceil\mathbf{a}P \in \mathbf{b}P \wedge x\lceil\mathbf{a}R \in \mathbf{b}R : x\},$$
$$\{x : x \in (\mathbf{a}P \cup \mathbf{a}R)^* \wedge x\lceil\mathbf{a}P \in \mathbf{t}P \wedge x\lceil\mathbf{a}R \in \mathbf{t}R : x\}.$$

The operator $||$, the connector, arises from trace theory (see van de Snepscheut [1985]), where it is called the weave operator. It is a shuffle, where common events occur simultaneously.

EXAMPLE 4.1. Consider the lock-free systems

$$P = \langle \{a, b, c, d\}, (abc|ad)\rangle, \qquad R = \langle \{a, c\}, (ac)\rangle.$$

Then we have

$$P\,||\,R = \langle \{a, b, c, d\}, \mathbf{pref}(ad|abc), (abc)\rangle.$$

For example, $abc \in \mathbf{t}(P\,||\,R)$ because $abc\lceil\mathbf{a}P = abc \in \mathbf{t}P$ and $abc\lceil\mathbf{a}R = ac \in \mathbf{t}R$.

EXAMPLE 4.2. For separate traces we can use the comma operator to denote weaving (notice that concatenation has higher priority when weaving and choice):

$$(ab\ ,\ bc) = (abc), \qquad (ab\ ,\ cb) = (acb|cab).$$

EXAMPLE 4.3. The operator $||$ denotes not only synchronization but also concurrency. Events that are not common may occur concurrently. If $x = ab$ and $y = de$, then $x\ ,\ y$ denotes the concurrent behavior of ab and de; i.e.,

$$x\ ,\ y = \{abde, adbe, adeb, dabe, daeb, deab\}.$$

Because of our assumption that any event occurs in infinitesimal time no two different events can really occur at the very same time, but always in some order. If that order is arbitrary, we have concurrency. In fact, the set x , y represents the sequences that can be observed outside the system. In trace theory (nor other language-based theories) any difference can be made between the concurrent behavior of events a and b and the choice between the sequential behaviors ab and ba. Both result in an observed behavior equal to a , b. Only in Petri nets (Peterson [1981]) this difference can be modeled.

In the supervisory theory of Ramadge and Wonham the above interaction of two systems is only part of the connection of a system (a plant) and its controller. In their approach the controller, dependent on the state he is in, can enable or disable events in the plant. Enabling or disabling in our approach is implicit. If the system is in a state where it is able to do some (common) event a and the controller is in a state where a is unable, a is disabled in the connection. Notice that this form of disabling events is fully symmetrical with respect to plant and controller. Moreover, we do not distinguish between controllable and uncontrollable events beforehand. Again, this is implicit. If we chose a controller R with alphabet $\mathbf{a}R$, then $\mathbf{a}R$ is the set of controllable events. However, $\mathbf{a}R$ may also contain events that are not in $\mathbf{a}P$. In our approach a controller is also a system. In the supervisory theory a controller is a function $f\colon \mathbf{b}P \to \Gamma$, giving, for each behavior of P a control input to be applied on P, by which events are disabled.

Our approach also deals with partial observations: only those events that are common to plant and controller can be observed by the controller. No additional observation alphabet and projection or mask is needed to model partial observability as has to be done in the supervisory approach (see Ramadge and Wonham [1989]).

4.2. Properties of the connection operator

The operator $\|$ defines a binary operation on the set of discrete event systems. It has some nice properties, which are listed below:

PROPERTY 4.1. For general systems P, R, and S, the following hold:

1. Symmetry: $P \| R = R \| P$.
2. Idempotency: $P \| P = P$.
3. Associativity: $(P \| R) \| S = P \| (R \| S)$.
4. The system **skip** $= \langle \emptyset, \{\epsilon\}, \{\epsilon\} \rangle$ is the unit element: $P \| \mathbf{skip} = P$.
5. The system **empty** $= \langle \emptyset, \emptyset, \emptyset \rangle$ is the zero element: $P \| \mathbf{empty} = \langle \mathbf{a}P, \emptyset, \emptyset \rangle$.

We do not have $P \| \mathbf{empty} = \mathbf{empty}$, so **empty** is not really a zero, but it reduces all behavior to nothing, so the name is not completely misplaced here.

DEFINITION 4.2. The ordering of discrete event systems is defined only for systems with equal alphabets as

$$P \subseteq R = (\mathbf{a}P = \mathbf{a}R \wedge \mathbf{t}P \subseteq \mathbf{t}R \wedge \mathbf{b}P \subseteq \mathbf{b}R).$$

P is called a *subsystem* of R.

PROPERTY 4.2. For discrete event systems P, R_1, and R_2 with $\mathbf{a}R_1 = \mathbf{a}R_2$, we have

$$R_1 \subseteq R_2 \Rightarrow (P \| R_1) \subseteq (P \| R_2).$$

Van de Snepscheut [1985] emphasized that it is essential to involve the alphabet into the definition of connection in order for $\|$ to be associative. Because of this associative property we can connect more discrete event systems without worrying about the order of computation. We use the following notation for connection of more discrete event systems (where $\mathcal{P}$ is some class of systems with equal alphabets):

$$(\| P : P \in \emptyset : P) = \mathbf{skip},$$

$$(\| P : P \in (R \cup \mathcal{P}) : P) = R \| (\| P : P \in \mathcal{P} : P).$$

4.3. A directed connection

Using the above definition of connection means dealing with synchronous interaction. Events have no direction: sending and receiving occurs at the same time. We also have *asynchronous interaction*: events then have a direction: sending goes before receiving. In that case inserting a coin by a person should come before accepting it by the vending machine and producing coffee should go before accepting it. In that case "sending" and "receiving" are different events (although related to each other). We can distinguish between sending (inserting a coin) and receiving (accepting it) by postfixing the event by ! and ? respectively. For example, *coin*! is inserting a coin, *coin*? is accepting it, *coffee*! is producing coffee, *coffee*? is accepting it. Asynchronous interaction now means that "receiving should come after sending," so *coffee*? should come after *coffee*!.

Asychronous interaction can be defined using the synchronous interaction operator $\|$. In order to do so we first divide the events into two kinds: inputs $\mathbf{i}P$ and outputs $\mathbf{o}P$, so that $\mathbf{a}P = \mathbf{i}P \cup \mathbf{o}P$ and $\mathbf{i}P \cap \mathbf{o}P = \emptyset$.

A (*directed*) *discrete event system* now is a discrete event system with all events in $\mathbf{o}P$ postfixed by ! and all events in $\mathbf{i}P$ postfixed by ?. Such a discrete event system is denoted by $P?!$. The (directed) task set of the vending machine equals

$$(coin? \cdot coffee!)^*,$$

and the (directed) task set of the person equals

$$(coin! \cdot coffee?)^*.$$

Connection can then be defined with the use of the following additional set and system:

$$\mathcal{T}(A) = \{a : a \in A : \langle \{a!, a?\}, (a! \cdot a?)^* \rangle\},$$

$$\mathbf{trans}(A) = (\| P : \mathrm{P} \in \mathcal{T}(A) : P).$$

$\mathcal{T}(A)$ is the class of all lock-free discrete event systems with alphabet equal to $\{a!, a?\}$ for some $a \in A$ and behavior equal to $(a! \cdot a?)^*$, i.e., first sending and then receiving. The system **trans**(A) is the shuffle (or parallel composition) of all these discrete event systems. **trans**(A) denotes the transmission of events from alphabet A: its task set equals all sequences in which for each event from A its sending part and its receiving part occurs equally often and alternatingly (first sending, then receiving). So

$$coin! \cdot coin? \cdot coffee! \cdot coin! \cdot coin? \cdot coffee?$$

belongs to the task set of **trans**$(\{coin,\ coffee\})$ but

$$coin! \cdot coin? \cdot coffee?$$

$$coin! \cdot coin! \cdot coffee! \cdot coin? \cdot coin?$$

do not (the first one, however, does belong to the behavior).

Definition 4.3. The directed connection of P and R is denoted by $P \overset{\leftrightarrow}{\|} R$ and defined only if $\mathbf{o}P \cap \mathbf{o}R = \mathbf{i}P \cap \mathbf{i}R = \emptyset$ by

$$P \overset{\leftrightarrow}{\|} R = P?! \;\|\; \mathbf{trans}(\mathbf{a}P \cap \mathbf{a}R) \;\|\; R?!.$$

Example 4.4. For our vending machine example (with V the vending machine and P the person) we find

$$\mathbf{t}P?! = (coin! \cdot coffee?)^*,$$

$$\mathbf{t}V?! = (coin? \cdot coffee!)^*,$$

$$\mathbf{t}(\mathbf{trans}(\{coin,\ coffee\})) = ((coin! \cdot coin?)^*\ ,\ (coffee! \cdot coffee?)^*),$$

which results in

$$\mathbf{t}(P \overset{\leftrightarrow}{\|} V) = (coin! \cdot coin? \cdot coffee! \cdot coffee?)^*.$$

In **trans** we have modeled that each output should first be followed by the corresponding input before that output may occur again (no insertion of a next coin if the previous one is not yet accepted). **trans**(A) is in fact some buffer: it may hold precisely one output event for each event in A (it is in fact a one place buffer for each event, see Example 3.3). It models *bounded delay* (each output should first be followed by the correspooonding input before it may occur again) and *overtaking* (different outputs are not necessarily followed by the corresponding inputs in the same order).

trans can also be defined in such a way that more outputs may take place before a corresponding input has occurred. If the number of occurrences of outputs that has not yet been received is infinite we speak of *unbounded delay.* In that case **trans** is defined using

$$\mathcal{T}(A) = \{a : a \in A : \langle \{a!, a?\}, \{x : x \in \{a!, a?\}^* \wedge (\forall y : y \leq x : y\mathrm{N}a! \geq y\mathrm{N}a?) : x\}\rangle\}.$$

An operator in trace theory exists that takes care of this unbounded delay. It is the *agglutinate* and can be found in Van de Snepscheut [1985]. Its disadvantage is its difficulty and (more important) it loses some important properties (regularity for example), which makes the operator unusable in state-space form.

5. State graph representation

As stated before, more representations exists for behavior and task set. In this section we will give an informal introduction to one possibility of representing discrete event systems as defined in this paper. For a much more detailed introduction we refer to Smedinga [1992a, b].

A well-known representation is using (finite) state automatons (Hopcroft and Ullman [1979]). As long as a system is realistic (prefix-closed behavior and each task a behavior) such automations can be used here. If we also want to have a automaton-like representation for GDESs we have to use extended automatons, using two kinds of final states:

Definition 5.1. A state graph is defined by $G = (A, Q, \delta, q, B, T)$ with

A	alphabet, finite set of symbols
Q	state set, possibly infinite
$\delta: Q \times A \to Q$	state transition function
$q \in Q$	initial state
$B \subseteq Q$	behavior states
$T \subseteq Q$	task states

A state in $Q \backslash (B \cup T)$ is called a *nonstate*.

The transition function δ defines paths through the graphs, i.e., $\delta(p, a)$ is the state, reachable from state p if event a occurs. We suppose δ is a total function, i.e., defined for all pairs $(p, a) \in Q \times A$. The function δ can be extended, which results in the *closure* of δ, given by δ^*: $Q \times A^* \to Q$ and defined by

$$\delta^*(p, \epsilon) = p,$$

$$\delta^*(p, ax) = \delta^*(\delta(p, a), x).$$

If $\delta^*(q, x) \in B$ the trace x belongs to the behavior of P and if $\delta^*(q, x) \in T$ the trace x belongs to the task set. Thus, a graph G represents the system

$$\mathbf{gds}(G) = \langle A, \{x : \delta^*(q, x) \in B : x\}, \{x : \delta^*(q, x) \in T : x\}\rangle.$$

A minimal graph representation of a system P can be found by collecting all equivalence classes of P according to the equivalence relation

$$x \mathbf{E} y = (\forall z : xz \in \mathbf{b}P = yz \in \mathbf{b}P \wedge xz \in \mathbf{t}P = yz \in \mathbf{t}R).$$

An equivalence class $[x] = \{y : x \mathbf{E}\, y : y\}$ of x is said to be a state in B if $x \in \mathbf{b}P$ and said to be a state in T if $x \in \mathbf{t}P$. This is an extension of the well-known way of finding an automaton, given some expression, e.g., see Hopcroft and Ullman [1979]. We refer to Smedinga [1992b] for more details.

Example 5.1. Consider the following, unrealistic system:

$$P = \langle \{a, b\}, (aa|ab), (a) \rangle .$$

The following equivalence classes can be found:

$$p_0 = [\epsilon] = \{\epsilon\},$$

$$p_1 = [a] = \{a\},$$

$$p_2 = [aa] = \{aa, ab\},$$

$$p_3 = [b] = \{a, b\}^* \backslash (p_0 \cup p_1 \cup p_2).$$

In Figure 1 the corresponding graph $G = (\{a, b\}, \{p_0, p_1, p_2, p_3\}, \delta, p_0, \{p_2\}, \{p_1\})$ is shown. In the diagram the initial state is denoted by an extra small arrow.

5.1. *State graph for the connection*

Once state graphs for P and R are given, we can construct a state graph for $P \parallel R$. The state graph for $P \parallel R$ is just the cartesian product of the state graphs of P and R.

Definition 5.2. Given $G_i = (A_i, Q_i, \delta_i, q_i, B_i, T_i)$ $(i = 1, 2)$ we define the *product graph* of G_1 and G_2 by

$$\mathbf{prod}(G_1, G_2) = (A_1 \cup A_2, Q_1 \times Q_2, \delta, (q_1, q_2), B_1 \times B_2, T_1 \times T_2),$$

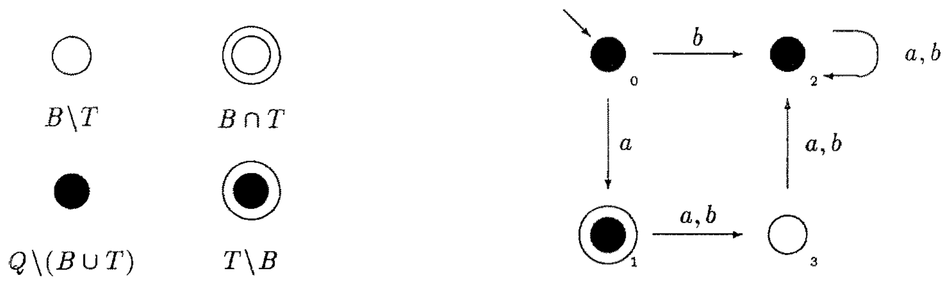

Figure 1. Left: displaying of different states; right: system from Example 5.1.

where

$$\delta((p_1, p_2), a) = \begin{cases} (\delta_1(p_1, a), p_2) & \text{if } a \in A_1 \backslash A_2, \\ (p_1, \delta_2(p_2, a)) & \text{if } a \in A_2 \backslash A_1, \\ (\delta_1(p_1, a), \delta_2(p_2, a)) & \text{if } a \in A_1 \cap A_2. \end{cases}$$

It should be clear that **prod**(G_1, G_2) represents **gds**$(G_1) \parallel$ **gds**(G_2).

For computing a state graph for $P \lceil A$ we need to implement alphabet restriction on state graphs. To get a state graph for $P \lceil A$, given a state graph for P, we can simply replace every transition labeled with a symbol not in A by a transition labeled with ϵ. However, in this way the result is a nondeterministic state graph. Standard techniques exists to get a deterministic state graph. In our extended version we have

DEFINITION 5.3. A nondeterministic state graph $G_{\text{nd}} = (A, Q, \gamma, q, B, T)_{\text{nd}}$ is defined with A, Q, q, B, and T as in Definition 5.2 and

$$\gamma \colon Q \times (A \cup \{\epsilon\}) \rightarrow 2^Q$$

the state transition map.

Again, γ can be extended to paths in the state graph (giving the closure γ^*). Such a path also consists of ϵ-transitions. Each path starting in q and ending in a state in B (T respectively) (possibly using some ϵ-transitions) represents a behavior (task respectively). An nd-graph can be made deterministic using the following extended standard technique:

$$\mathbf{det}(G_{\text{nd}}) = (A, 2^Q, \delta, \bar{q}, \bar{B}, \bar{T}),$$

where (for $r \in 2^Q$)

$$\delta(r, a) = \bigcup_{p \in r} \gamma^*(p, a),$$

$$\bar{q} = \gamma^*(q, \epsilon),$$

$$\bar{B} = \{r : r \cap B \neq \emptyset : r\},$$

$$\bar{T} = \{r : r \cap T \neq \emptyset : r\}.$$

Notice that each state in **det**(G_{nd}) is a set of states of G_{nd}.

The above construction is a generalization of the well-known construction to find the deterministic equivalent of a nondeterministic automaton. Apart from unreachable states, each state in **det**(G_{nd}) is the set of states that can be reached from another set by doing zero or more ϵ-moves, followed by one normal move, followed by zero or more ϵ-moves. Notice that the above construction also holds for graphs with infinitely many states.

It should be clear that $\mathbf{det}(G_{\mathrm{nd}})$ represents the same system as G_{nd} but is not deterministic. This leads to a way of finding a state graph for the external connection defined by $P \rceil\lceil R = (P \| R) \lceil (\mathbf{a}P \div \mathbf{a}R)$, given state graphs G_1 for P and G_2 for R. First compute $\mathbf{prod}(G_1, G_2)$ using definition 5.2, then replace every transition in this state graph labeled with an event not in $\mathbf{a}P \div \mathbf{a}R$ by ϵ and compute, using the above construction, the deterministic equivalent.

EXAMPLE 5.2. Consider the state graphs as given in Figure 2a, b. The connection is given in Figure 2c. Notice that this state graph is not minimal: all nonstates can be replaced by one nonstate. We also have computed the external connection, see Figure 2d, using this minimized graph and made this state graph deterministic again, see Figure 2e. In this last figure, only the reachable states are given. Notice that P and R are one-place buffers and their external connection is a two-place buffer.

5.2. *Regular systems*

We say a system P is regular if there exists a finite state graph representation $\mathbf{gds}(P)$ for it. It is easily seen that if the state graphs for P and R have a finite number of states, so do the constructed state graphs for $P \| R$, $P \lceil A$, and $P \rceil\lceil R$, meaning that if P and R are regular, so are $P \| R$, $P \lceil A$, and $P \rceil\lceil R$.

6. Locked systems

Our definition of a discrete event system makes it easy to define the notion of lock. Each behavior of P from which it is impossible to complete a task is part of the locked behavior. A system P is said to be locked if no task can be completed any more.

DEFINITION 6.1. For a discrete event system P we define its set of *locked* traces, denoted by $\mathbf{lock}(P)$, by

$$\mathbf{lock}(P) = \{x : x \in \mathbf{b}P \wedge (\forall y : y \in (\mathbf{a}P)^* : xy \notin \mathbf{t}P) : x\}.$$

A discrete event system P is *lock-free* if $\mathbf{lock}(P) = \emptyset$.

The set of locked traces consists of those behaviors that cannot be completed. Therefore, the locked traces are those traces that are in $\mathbf{b}P$ and not in $\mathbf{pref}(\mathbf{t}P)$, so:

PROPERTY 6.1.

$$\mathbf{lock}(P) = \mathbf{b}P \backslash \mathbf{pref}(\mathbf{t}P).$$

EXAMPLE 6.1. Reconsider Example 4.1. Then we have

$$\mathbf{lock}(P \| R) = \{ad\}.$$

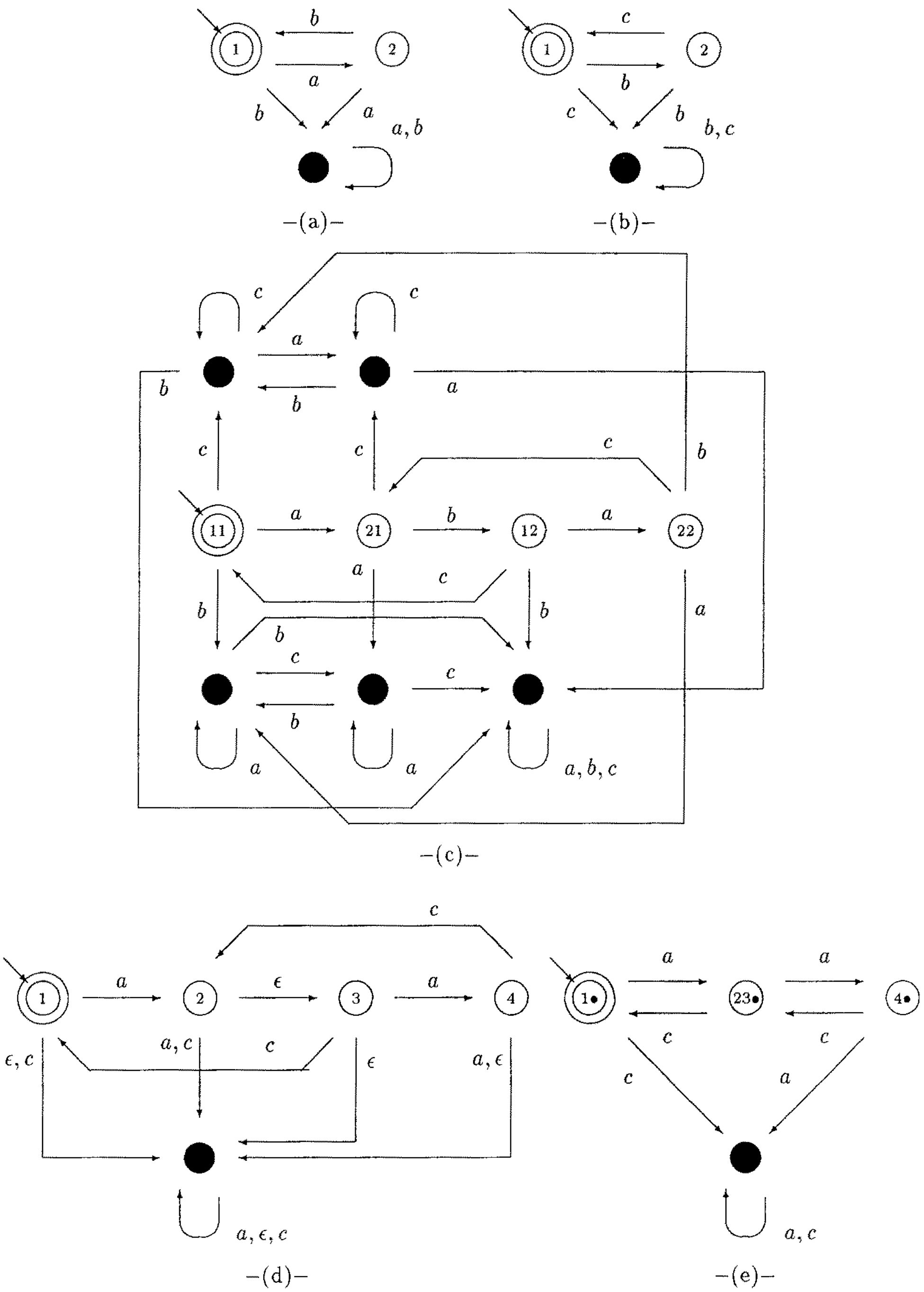

Figure 2. Graphs for Example 5.2: (a) state graph G_1; (b) state graph G_2; (c) product graph of G_1 and G_2; (d) nondeterministic graph for external connection of **gds**(G_1) and **gds**(G_2); (e) its deterministic equivalent.

So the connection of P and R may lock, i.e., if first the common event a occurs and next P performs event d then P has completed a task but R has not. The only remaining possibility for R is to perform c but P does not allow this. The connection is locked.

In fact, the set **lock**(P) as defined above, contains two kinds of locked behavior. A system can deadlock (unable to perform a next event in some behavior that is not a completed task); e.g.,

$$\mathbf{deadlock}(P) = \{x : x \in \mathbf{b}P \wedge x \notin \mathbf{t}P : (\forall a : a \in \mathbf{a}P : xa \notin \mathbf{b}P) : x\},$$

or it can livelock (a next event is always possible, although a completed task will never be reached); e.g.,

$$\begin{aligned}\mathbf{livelock}(P) = \{x : x \in \mathbf{b}P \wedge &\\ &(\forall y : y \in (\mathbf{a}P)^* \\ &: xy \notin \mathbf{t}P \wedge (\exists a : \mathbf{a} \in \mathbf{a}P : xya \in \mathbf{b}P)) : x\}.\end{aligned}$$

Notice that

$$\mathbf{deadlock}(P) \subseteq \mathbf{lock}(P),$$

$$\mathbf{livelock}(P) \subseteq \mathbf{lock}(P).$$

Our definition includes both cases and also all behavior that can only result in deadlock, but is itself not yet deadlocked.

The term *deadlock* corresponds to the accepted meaning of the word in computing science: A system is deadlocked if it is impossible to continue. The term *livelock* is perhaps less common. It does not mean that certain subbehavior can be repeated for ever and ever, but in addition that, although there is no deadlock, no task can be completed.

In Kaldewaij [1988] an alternative definition of (dead)lock is given, saying that a connection of systems has the property to (dead)lock if one of the subsystems has that property. In that case no controller can be found to give a (dead)lock-free connection. In our approach a connection of systems has the possibility to (dead)lock if it is possible to reach a behavior *in that connection* from which no task can be completed. Even if each of the subsystems has the possibility to lock then the connection can be lock-free!

It is possible to get rid of the lock by adding a second system as is shown in the following example.

EXAMPLE 6.2. Consider

$$P = \langle \{a, b, c, d\}, \mathbf{pref}(ad|abc), (abc)\rangle.$$

Then **lock**$(P) = \{ad\}$. If we connect this system with $R = \langle \{b, d\}, (b)\rangle$ we find the connection

$$P \,\|\, R = \langle \{a, b, c, d|, (abc)\rangle$$

that is free of lock. Notice that $R = \langle \{b\}, (b)\rangle$ leads to $P \,\|\, R = P$ so $\langle \{b\}, (b)\rangle$ does not remove the lock.

7. Lock-free subsystems

It is straightforward to find the greatest subsystem of some given system P that is free of lock. It simply is the system

$$\langle \mathbf{a}P,\ \mathbf{b}P \cap \mathbf{pref}(\mathbf{t}P),\ \mathbf{t}P \cap \mathbf{b}P\rangle.$$

Just delete all behavior that does not lead to a completed task and delete all tasks that have no proper behavior leading to that task.

8. Lock-free connections

We claim that, if some arbitrary P and R have (in connection) the possibility of lock, we can construct a subsystem, say S, of R such that $\mathbf{lock}(P \parallel S) = \emptyset$, where $S \sqsubseteq R$. Of course, the system **empty** will always satisfy. However, we are searching for the largest possible S.

Notice that we do not impose any restrictions on the alphabets of P and R. In most cases, however, R controls P by using a subset of the events of P. The theory below assumes $\mathbf{a}P$ and $\mathbf{a}R$ to be arbitrary.

EXAMPLE 8.1. First, notice that S itself need not be lock-free in order to obtain a lock free connection with P. Consider P and S given by

$$P = \langle \{a, b, c\},\ \mathbf{pref}(ab),\ (a)\rangle,$$

$$S = \langle \{a, b, c\},\ \mathbf{pref}(ac),\ (a)\rangle.$$

Then

$$P \parallel S = \langle \{a, b, c\},\ \mathbf{pref}(a),\ (a)\rangle$$

is lock-free, but neither one of P and S is.

In the sequel we try to find an algorithm that deletes some traces from the behavior and task set of a discrete system in order to get a lock-free connection with a second system.

Simply deleting traces from the behavior and task set may lead to a system that is no longer a DES. Therefore, we introduce an operator on GDESs to get the so called DES interior of it, the greatest subsystem that is a real DES, i.e., satisfies the properties of having a prefix-closed behavior and a task set that is part of the behavior.

DEFINITION 8.1. If P is some *GDES* we define the *DES interior of P*, denoted by $\mathbf{des}(P)$, by

$$\begin{aligned}\mathbf{des}(P) = \langle \mathbf{a}P,\ &\{x : x \in \mathbf{b}P \wedge (\forall y : y \leq x : y \in \mathbf{b}P) : x\},\\ &\{x : x \in \mathbf{t}P \wedge (\forall y : y \leq x : y \in \mathbf{b}P) : x\}\rangle.\end{aligned}$$

If P is a discrete event system and T is some set of traces over alphabet $\mathbf{a}P$, then we define P without T as the system

$$P \setminus\!\setminus T = \mathbf{des}(P \setminus T),$$

where $P \setminus T = \langle \mathbf{a}P, \mathbf{b}P \setminus T, \mathbf{t}P \setminus T \rangle$.

Notice that $P \setminus\!\setminus T$ is a well-defined DES, i.e., has a prefix-closed behavior and each completed task is also a behavior.

EXAMPLE 8.2. If $P = \langle \{a, b, d\}, \mathbf{pref}(|b|aba), (a|aba) \rangle$ and $T = \{ab\}$, then we find

$$P \setminus T = \langle \{a, b, d\}, (\epsilon|a|b|aba), (a|aba) \rangle,$$

$$P \setminus\!\setminus T = \langle \{a, b, d\}, (\epsilon|a|b), a \rangle.$$

If P itself is a well-defined DES (i.e., with prefix-closed behavior and each task a behavior), we can define the operator $P \setminus\!\setminus T$ directly, without using the DES interior, by

$$P \setminus\!\setminus T = \langle \mathbf{a}P, \mathbf{b}P \setminus (T(\mathbf{a}P)^*), \mathbf{t}R \setminus (T(\mathbf{a}P)^*) \rangle.$$

It should be clear that this definition is equivalent to the previous one: instead of deleting the traces from T and creating holes in the trace sets (which are removed by computing the DES interior afterwards), you can also delete all traces form T together with all their extensions.

EXAMPLE 8.2. (cont.)

$$T(\mathbf{a}P)^* = (ab(a|b|d)^*) = \{ab, aba, abd, abaa, abab, \ldots\}.$$

Deleting this trace set from $\mathbf{b}P$ and $\mathbf{t}P$ directly gives the same result.

EXAMPLE 8.3. Simply deleting all locked traces from one of the systems P and R using this new operator will not be enough to find a subsystem of R that has a lock-free connection with P. Consider

$$P = \langle \{a, b, c\}, (aca|aaba) \rangle, \qquad R = \langle \{a\}, (a|aa) \rangle.$$

Then we have

$$P \parallel R = \langle \{a, b, c\}, \mathbf{pref}(aab|aca), (aca) \rangle,$$

so $\mathbf{lock}(P \parallel R) = \{aa, aab\}$. Computing

$$S = R \setminus\!\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$$

results in $S = \langle \{a\}, (a), \emptyset \rangle$ and

$$P \parallel S = \langle \{a, b, c,\}, \mathbf{pref}(ac), \emptyset \rangle,$$

which is, again, not free of lock.

From now on we assume that we have well-defined DESs P and R and are searching for the greatest subsystem of R that has a lock-free connection with P.

Our claim is that the following operator leads to the greatest possible lock-free subsystem:

$$L(P, R) = R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R.$$

Starting with $R_0 = R$ we can compute $R_{i+1} = L(P, R_i)$ and so find a chain of R_i's. We claim that the operator L has a fixpoint and that this fixpoint is the subsystem of R we are looking for.

The first property says that each iteration reduces the resulting subsystem and gives an important property of **lock**.

Property 8.1

(a) $L(P, R) \subseteq R$.
(b) $\mathbf{lock}(P \parallel R) \lceil \mathbf{a}R \subseteq \mathbf{b}R$.

If the fixpoint exists it leads to a lock-free connection with P:

Property 8.2

$$\mathbf{lock}(P \parallel S) = \emptyset \Leftrightarrow L(P, S) = S.$$

Proof

$$L(P, S) = S$$

$$= \mathbf{des}(S \setminus \mathbf{lock}(P \parallel S) \lceil \mathbf{a}S) = S \qquad \text{[definition of operator } L]$$

$$= \mathbf{lock}(P \parallel S) \lceil \mathbf{a}S = \emptyset \qquad [\mathbf{lock}(P \parallel S) \lceil \mathbf{a}S \subseteq \mathbf{b}S].$$

Lemma 8.1

$$(L(P, S) = S \wedge S \subseteq R) \Rightarrow S \subseteq L(P, R).$$

Proof

$$\mathbf{b}S \subseteq \mathbf{b}(R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R)$$

$$= \mathbf{b}S \subseteq \mathbf{b}R \setminus (\mathbf{lock}(P \parallel R) \lceil \mathbf{a}R)(\mathbf{a}R)^* \qquad \text{[second definition of } \setminus\setminus]$$

$= (\forall y, z : y \in \mathbf{lock}(P \parallel R) \wedge z \in (\mathbf{a}R)^* : (y \lceil \mathbf{a}R)z \notin \mathbf{b}S)$ $\quad [S \subseteq R]$

$= (\forall y, z : y \in \mathbf{b}(P \parallel R) \wedge y \in \mathbf{lock}(P \parallel R) \wedge z \in (\mathbf{a}R)^* : (y \lceil \mathbf{a}R)z \notin \mathbf{b}S)$
$[\mathbf{lock}(P) \subseteq \mathbf{b}P]$

$= (\forall y, z : y \in \mathbf{b}(P \parallel R) : z \notin (\mathbf{a}R)^* \vee (y \lceil \mathbf{a}R)z \notin \mathbf{b}S \vee y \notin \mathbf{lock}(P \parallel R))$ $\quad$ [trading]

$= (\forall y : y \in \mathbf{b}(P \parallel R) : (\forall z :: z \notin (\mathbf{a}R)^* \vee (y \lceil \mathbf{a}R)z \notin \mathbf{b}S \vee y \notin \mathbf{lock}(P \parallel R)))$
[nesting]

$= (\forall y : y \in \mathbf{b}(P \parallel R) : y \notin \mathbf{lock}(P \parallel R)) \vee (\forall z :: z \notin (\mathbf{a}R)^* \vee (y \lceil \mathbf{a}R)z \notin \mathbf{b}S)$
[distribution of $\vee$ over $\forall$]

$= \forall y : y \in \mathbf{b}(P \parallel R) : y \notin \mathbf{lock}(P \parallel R)) \vee \neg(\exists z :: z \in (\mathbf{a}R)^* \wedge (y \lceil \mathbf{a}R)z \in \mathbf{b}S)$
[deMorgan]

$= (\forall y : y \in \mathbf{b}(P \parallel R) \wedge (\exists z :: z \in (\mathbf{a}R)^* \wedge (y \lceil \mathbf{a}R)z \in \mathbf{b}S) : y \notin \mathbf{lock}(P \parallel R))$
[trading]

$= (\forall y : y \in \mathbf{b}(P \parallel R) \wedge y \lceil \mathbf{a}R \in \mathbf{b}S : y \notin \mathbf{lock}(P \parallel R))$ $\quad$ [$\mathbf{b}S$ is prefix closed]

$= (\forall y : y \in \mathbf{b}(P \parallel S) : (\exists w :: yw \in \mathbf{t}(P \parallel R)))$
[$S \subseteq R$, and definitions of $\parallel$ and **lock**]

$\Leftarrow (\forall y : y \in \mathbf{b}(P \parallel S) : (\exists w :: yw \in \mathbf{t}(P \parallel S)))$ $\quad [S \subseteq R]$

$=$ true $\quad$ [$(P \parallel S)$ is lock free]

On the other hand:

$\mathbf{t}S \subseteq \mathbf{t}(R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R)$

$= \mathbf{t}S \subseteq \mathbf{t}(R \backslash (\mathbf{lock}(P \parallel R) \lceil \mathbf{a}R)(\mathbf{a}R)^*$ $\quad$ [second definition of $\setminus\setminus$]

$\Leftarrow \mathbf{t}S \subseteq \mathbf{t}R \wedge \mathbf{t}S \subseteq \mathbf{b}S$ $\quad$ [above result and R is a DES]

$=$ true $\quad$ [$S \subseteq R$ and S is a DES].

This combines to $S \subseteq R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$. Notice that we have used here that the behavior of both S and R is prefix-closed and each task is a behavior.

We define the following set of classes of DESs:

$$\Phi = \{\,\mathcal{V} : R \in \mathcal{V} \wedge (\forall S : S \in \mathcal{V} : L(P, S) \in \mathcal{V})$$
$$\wedge(\forall \mathcal{U} : \breve{\ } \subseteq \mathcal{V} : \bigcap_{S \in \mathcal{U}} S \in \mathcal{V}) : \mathcal{V}\}.$$

The set

$$\{S : S \subseteq R : S\} \in \Phi$$

is not empty because

(a) $R \in \{S : S \subseteq R : S\}$
(b) $(\forall S : S \in R : L(P, S) \subseteq R)$ (see Property 8.1 (a)).
(c) $(\forall \mathcal{U} : \mathcal{U} \subseteq \{S : S \subseteq R : S\} : \cap_{S \in \mathcal{U}} S \subseteq R)$.

Next, we define the intersection of all $\mathcal{V}$ from Φ to be the set $\mathcal{W}$:

$$\mathcal{W} = \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}.$$

We now have that this $\mathcal{W}$ is also a member of the class Φ:

PROPERTY 8.3

(a) $R \in \mathcal{W}$.
(b) $(\forall S : S \in \mathcal{W} : L(P, S) \in \mathcal{W})$.
(c) $(\forall \mathcal{U} : \mathcal{U} \subseteq \mathcal{W} : \cap_{S \in \mathcal{U}} S \in \mathcal{W})$.

Proof

(a) true

$$= (\forall \mathcal{V} : \mathcal{V} \in \Phi : R \in \mathcal{V}) \qquad \text{[definition of } \Phi\text{]}$$

$$= R \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}.$$

(b) $S \in \mathcal{W}$

$$= S \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V} \qquad \text{[definition of } \mathcal{W}\text{]}$$

$$= (\forall \mathcal{V} : \mathcal{V} \in \Phi : S \in \mathcal{V})$$

$\Rightarrow (\forall \mathcal{V} : \mathcal{V} \in \Phi : L(P, S) \in \mathcal{V})$ [definition of Φ, second property of $\mathcal{V}$]

$= L(P, S) \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}$

$= L(P, S) \in \mathcal{W}$ [definition of $\mathcal{W}$]

(c) $\mathcal{U} \subseteq \mathcal{W}$

$= \mathcal{U} \subseteq \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}$ [definition of $\mathcal{W}$]

$= (\forall \mathcal{V} : \mathcal{V} \in \Phi : \mathcal{U} \subseteq \mathcal{V})$

$\Rightarrow (\forall \mathcal{V} : \mathcal{V} \in \Phi : \bigcap_{S \in \mathcal{U}} S \in \mathcal{V})$ [definition of Φ, third property of $\mathcal{V}$]

$= \bigcap_{S \in \mathcal{U}} S \in \bigcap_{\mathcal{V} \in \Phi} \mathcal{V}$

$= \bigcap_{S \in \mathcal{W}} S \in \mathcal{W}$ [definition of $\mathcal{W}$].

Now define

$$\Lambda(P, R) = \bigcap_{S \in \mathcal{W}} S.$$

Then we have that $\Lambda(P, R)$ is a fixpoint of the operator L.

PROPERTY 8.4

$$\Lambda(P, R) = L(P, \Lambda(P, R)).$$

Proof. By construction of L we have $L(P, \Lambda(P, R)) \subseteq \Lambda(P, R)$; see Property 8.1. Moreover,

$$\Lambda(P, R) = \bigcap_{S \in}$$

$\Rightarrow \Lambda(P, R) \in \mathcal{W}$ [see Property 8.3(c) with $\mathcal{U} = \mathcal{W}$]

$\Rightarrow L(P, \Lambda(P, R)) \in \mathcal{W}$ [see Property 8.3(b)].

Hence,

$$\begin{aligned} & \Lambda(P, R) \\ = & \bigcap_{S \in \mathcal{W}} S \quad \text{[definition]} \\ \subseteq & L(P, \Lambda(P, R)) \quad [L(P, \Lambda(P, R)) \in \mathcal{W}]. \end{aligned}$$

The fixpoint $\Lambda(P, R)$ leads to a lock-free connection.

LEMMA 8.2

$$\mathbf{lock}(P \parallel \Lambda(P, R)) = \emptyset.$$

Proof. From Property 8.2 with $S = \Lambda(P, R)$ and property 8.4.

Moreover, we have that $\Lambda(P, R)$ is the greatest fixpoint, so it is the first one to be reached while computing the chain R_i.

LEMMA 8.3

$$(S = L(P, S) \wedge S \subseteq R) \Rightarrow S \subseteq \Lambda(P, R).$$

Proof. Let $\mathcal{V} = \{T : S \subseteq T \subseteq R : T\}$, then we have

(a) $R \in \mathcal{V}$,

(b) $T \in \mathcal{V}$

$\Rightarrow S \subseteq T \subseteq R$ [definition of $\mathcal{V}$]

$\Rightarrow S \subseteq L(P, T) \subseteq R$ [Lemma 8.1 with $L(P, S) = S$ and $L(P, T) \subseteq T$]

$\Rightarrow L(P, T) \in \mathcal{V}$.

(c) $\mathcal{U} \subseteq \mathcal{V}$

$\Rightarrow (\forall T : T \in \mathcal{U} : T \in \mathcal{V})$

$\Rightarrow \bigcap_{T \in \mathcal{U}} T \in \mathcal{V}$.

(a), (b), and (c) lead to $\mathcal{V} \in \Phi$, and hence, by definition of $\mathcal{W}$, that $\mathcal{W} \subseteq \mathcal{V}$. Now we have

$$\begin{aligned} & \Lambda(P, R) \\ = & \bigcap_{S \in \mathcal{W}} S \quad \text{[definition]} \\ \in & \mathcal{W} \quad [\text{Property 30(c) with } \mathcal{U} = \mathcal{W}] \\ \subseteq & \mathcal{V} \quad \text{[above]}. \end{aligned}$$

So $\Lambda(P, R) \in \mathcal{V}$; that is, $S \subseteq \Lambda(P, R) \subseteq R$, so $\Lambda(P, R)$ is the biggest one.

We now have proved the following theorem.

THEOREM 8.1. $\Lambda(P, R)$ is the largest system contained in R for which the connection with P is free of lock.

EXAMPLE 8.4. If the fixpoint is empty, no subsystem of R can be found such that the connection with P is free of lock. Consider P and R from Example 8.3. Then

$$R_0 = R,$$

$$R_1 = \langle \{a\}, (a \mid aa) \backslash (aab \mid aa) \rangle = \langle \{a\}, (a) \rangle,$$

$$R_2 = \langle \{a\}, (a) \backslash (a \mid ac) \rangle = \langle \{a\}, \emptyset \rangle,$$

and, indeed, no nonempty subsystem of R can be found with a lock-free connection (just try every possible subsystem of R).

8.1. Deadlock-free connections

Actually, we can prove the same for the deadlock case. Use

$$\Delta(P, R) = \bigcap_i R_i \quad \text{with} \begin{cases} R_0 = R, \\ R_i = R_{i-1} \mathbin{\backslash\backslash} \mathbf{deadlock}(P \parallel R_{i-1}) \lceil \mathbf{a}R, \quad i = 1, 2, \ldots. \end{cases}$$

Then we have

THEOREM 8.2. $\Delta(P, R)$ is the largest system contained in R for which the connection with P is free of deadlock.

9. Effectively computable

Using the state graphs defined earlier, we are able to compute $\Lambda(P, R)$ effectively. It remains to find the set $\mathbf{lock}(P)$ of some given P using state graphs and to compute $P \mathbin{\backslash\backslash} T$ for some system P and trace set T using a state graph representation of P and T.

9.1. Finding locked states

If $x \in \mathbf{lock}(P)$ then the state $\delta^*(q, x)$, in some state graph G representing P, has the property that no path, starting in this state ever reaches a task state. Such states can easily be found by using the reversed graph of G, i.e., with all transitions reversed. In this graph each state that is reachable from a task state is a state in the original graph from which a task state can be reached. Each behavior state that cannot be reached from a task state in the reversed graph, therefore, is a locked state in the original graph.

Once we have found the locked states we can construct a graph that represents the system $\langle \mathbf{a}P, \mathbf{lock}(P), \mathbf{lock}(P)\rangle$ by changing all states in nonstates and all locked states in behavior and task states.

Example 9.1. Consider the system P with representation as given in Figure 3 (left). In the figure we have omitted all transitions going to the only nonstate of the representation. The right part of the figure shows the reversed graph. From the task state 9 only the states 1 and 2 can be reached. This means that the states 3, 4, 5, 6, 7, and 8 are locked states. $\mathbf{lock}(P)$ can now be represented using the left graph of the figure, where states 1, 2, and 9 are made nonstates and states 3, 4, 5, 6, 7, and 8 are made behavior and task states.

9.2. *The minus graph*

The following algorithm can be used to compute $P \backslash\backslash T$, given graph G_1 for P and G_2 for T.

Definition 9.1. Associated with two state graphs $G_1 = (A, Q_1, \delta_1, q_1, B_1, T_1)$ and $G_2 = (A, Q_2, \delta_2, q_2, B_2, T_2)$ we define the *minus graph* $\mathbf{min}(G_1, G_2)$ by

$$\begin{array}{ll} (A, \{(q_1, q_2)\}, 1, (q_1, q_2), \emptyset, \emptyset) & \text{if } q_2 \in T_2, \\ (A, Q_1 \times Q_2, \delta, (q_1, q_2), B_1 \times (Q_2 \backslash B_2), T_1 \times (Q_2 \backslash T_2)) & \text{otherwise,} \end{array}$$

where, for $p_1 \in S_1$, $p_2 \in S_2$, and $a \in A$, d is defined by

$$\delta((p_1, p_2), a) = (d_1(p_1, a), d_2(p_2, a)).$$

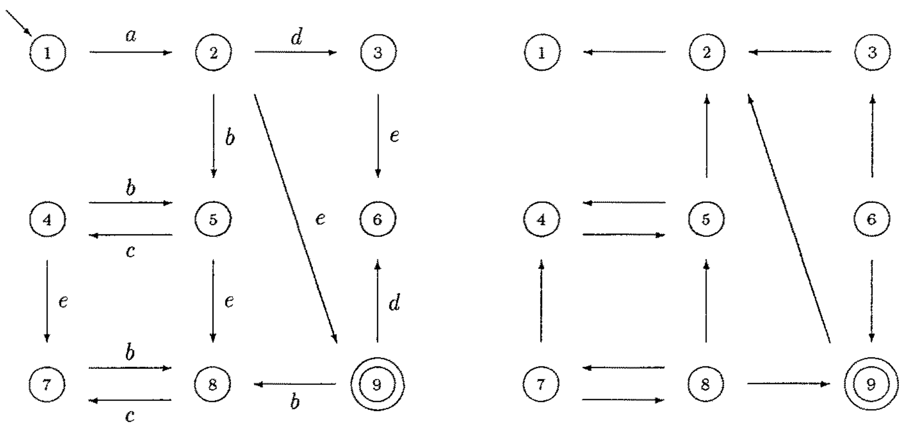

Figure 3. State graph of system for Examle 9.1 (left) and its reverse graph (right).

Again we use a cartesian product of two graphs, but this time behavior and task states are constructed differently. It should be clear that the graph **min**(G_1, G_2) represents $P \setminus\setminus T$.

EXAMPLE 9.2. Suppose P and R are given as in Figure 4a, b. The product graph, representing the connection, can be found in Figure 3 (left). From the previous example and the above construction we find a graph for **lock**$(P \parallel R)$ by making the states 1, 2, and 9 in

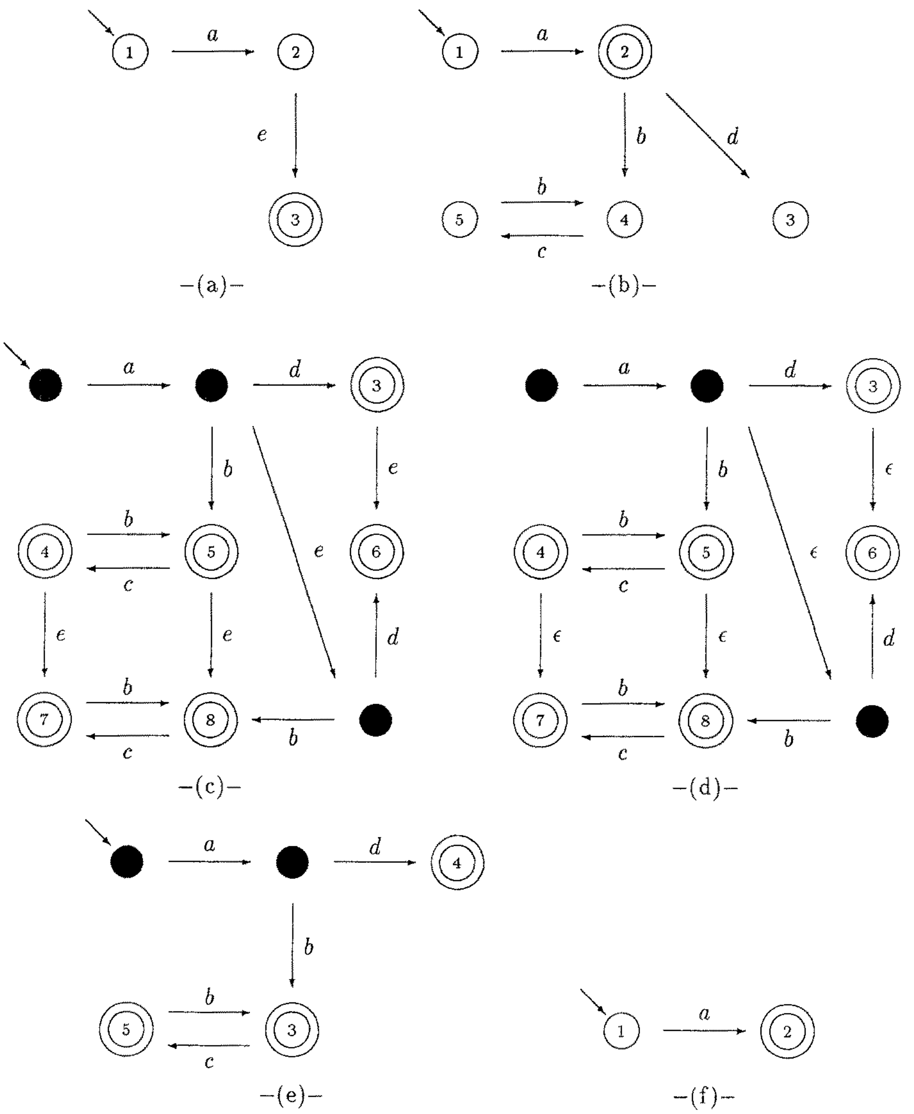

Figure 4. Graphs for Example 9.2: (a) state graph for system P; (b) state graph for system R; (c) state graph for **lock**$(P \parallel R)$; (d) nondeterministic state graph for **lock**$(P \parallel R) \lceil$aR; (e) deterministic graph for **lock**$(P \parallel R) \lceil$aR; (f) graph for $R \setminus\setminus$ **lock**$(P \parallel R) \lceil$aR.

this graph nonstates and making the other states behavior and task states (Figure 4c). Next, we replace every symbol not in $\mathbf{a}R$ by ϵ to get an nd-graph for $\mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$ (Figure 4d). The deterministic equivalent of this graph can be found in Figure 4e. The minus graph of the graphs of R and the last graph results in graph as displayed in Figure 4f, representing $R \setminus\setminus \mathbf{lock}(P \parallel R) \lceil \mathbf{a}R$.

10. A comparison with other approaches

Also in the framework of Ramadge and Wonham there exists theory in finding so-called nonblocking supervisors, i.e., a supervisor that not only leads to the desired behavior but also to a lock-free connection (see Li and Wonham [1988] and Ramadge and Wonham [1989]).

The nonblocking property is an extra demand on the supervisor. In our approach lock-free (or nonblocking) is a demand on its own. Given some plant P we can construct a controller R that leads in connection with P to a lock-free system. Our approach is more general than finding a nonblocking supervisor because (*a*) no restrictions are put on the system P that should be controlled (we only need that P has a realistic interpretation, i.e., $\mathbf{pref}(\mathbf{b}P) = \mathbf{b}P$ and $\mathbf{t}P \subseteq \mathbf{b}P$), especially no modeling of the behavior in some way is necessary (no automatons), and (*b*) getting a lock-free connection is a demand on its own (and not part of another control problem like with the nonblocking supervisor).

The method discussed above can also be used as a second step in case a controller has to be found such that the connection of controller and plant meets certain constraints (for example, $P \parallel R$ should be within certain minimal and maximal behaviors). First controller R can be computed such that the constraints are met (without worrying about lock) and afterwards from R a new controller can be derived that also leads to a lock-free connection. If the connection still satisfies the needed constraints we have found a solution, if not, no solution is possible (provided the original controller R was as large as possible). In Smedinga [1991] more can be found about this approach.

Our approach leads to a number of related problems, for example we might try to find (in some sense) the minimum set of events that have to be controlled (i.e., the minimum set $\mathbf{a}R$) to get a lock-free connection.

11. An example

We end with an illustration of the results. Reconsider the system of the farmer, wolf, goat, and cabbage. the following events are possible:

f	farmer goes to other side alone
w	farmer takes wolf to other side
g	farmer takes goat to other side
c	farmer takes cabbage to other side
e	one thing is eaten by another

We give the behavior and task set of this system by using the graph of Figure 5. The states, the system can be in, can be denoted by $(ijkl)$ with $i, j, k, l = 1, 2$, where i is the position

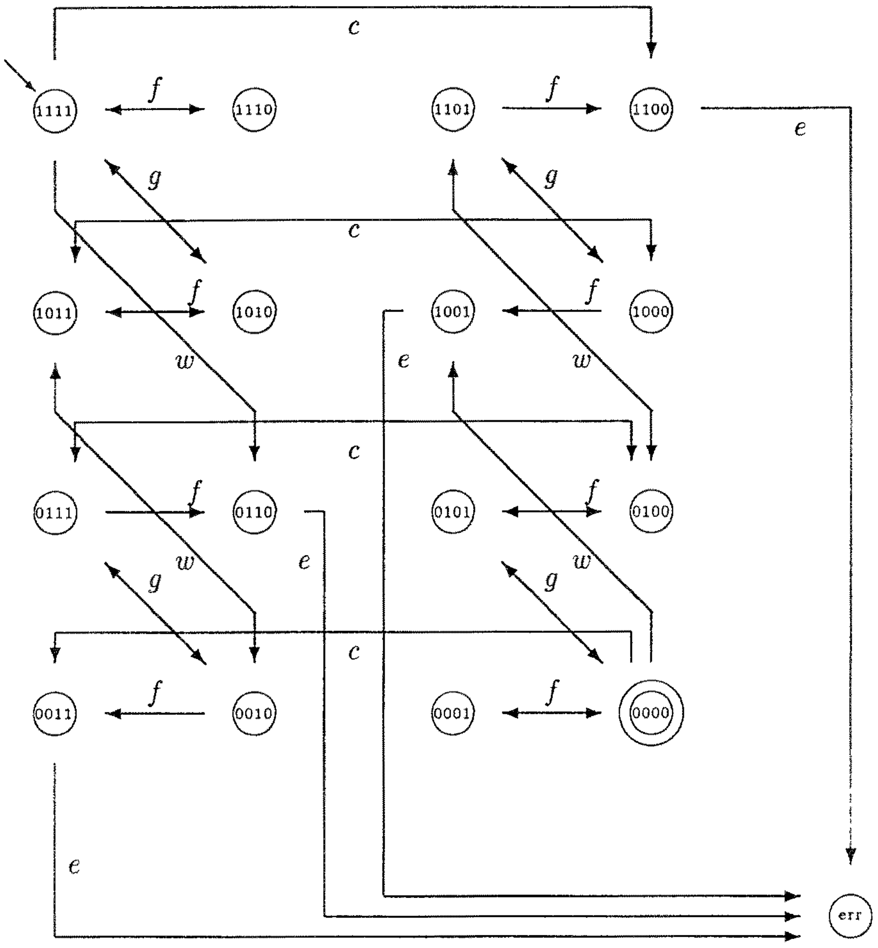

Figure 5. Behavior of farmer system.

of the wolf, *j* the position of the goat, *k* the position of the cabbage, *l* the position of the farmer, where 0 means on this side of the river and 1 means on the other side. For example state (1011) means that wolf, cabbage, and farmer are on the other side and the goat is on this side of the river. After one thing is eaten the positions are not important anymore (the farmer simply did not succeed), so the state after eating is denoted by (err). In the figure we omit the nonstates and every transition going to it.

We start in state (1111), i.e., on the other side of the river. According to the puzzle the farmer should bring wolf, goat, and cabbage to this side of the river, i.e., a legal ending is in state (0000). Each behavior, starting in (1111) and ending in (0000) therefore is a task of the system.

In Figure 5 all possible behaviors of the farmer system are denoted. Notice that it is possible to reach state (0000) from each other state, except from state (err). All behavior ending in (err) therefore is locked behavior. We want to get rid of this behavior by controlling the behavior of the farmer, i.e., control events w, g, c, and f in such a way that it becomes impossible to lock (notice that occurrence of e is the only possibility for lock). Event e cannot be controlled so we cannot give a controller that simply forbids event e to happen.

In this example we are free to chose system R from which we compute the largest subsystem that leads to a lock-free controller. Because we want to have a minimal restrictive controller we start with a system r that is as large as possible:

$$R = \langle \{f, w, g, c\}, (f|w|g|c)^* \rangle .$$

In general, this is the way to get a minimal restrictive controller: start with the largest one that is possible, i.e., the most general controller that can be found and use the above theory to compute the largest subsystem of it with the required properties. In this way we impose as less restrictions on the original system P as possible, ending up with a system with as much freedom as is possible provided no lock can occur any more (see Smedinga [1991]).

Now we compute $\Lambda(P, R)$. A computer program, using algorithms based on the state graph representations as presented before, is used to compute the sequence R_i, resulting in $R_1 = \Lambda(P, R)$. Computing the connection $P \parallel R_1$ leads to the system as is displayed in Figure 6. The event e never occurs; there is no danger of lock.

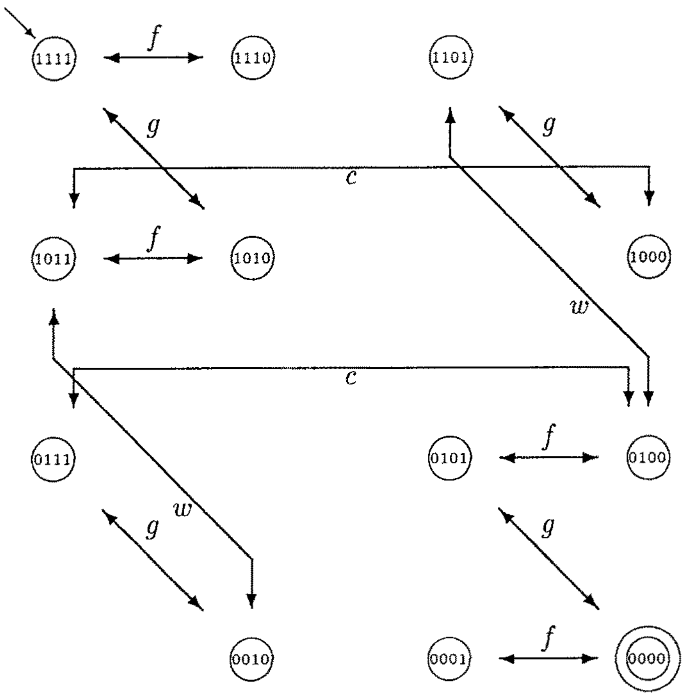

Figure 6. Controlled behavior of farmer system.

If we do not consider unnecessary loops, in fact two possible behaviors result:

$gfwgcfg$,
$gfcgwfg$,

which are precisely the two possible solutions of the puzzle.

It is even possible to consider only events f, w, and c to be controllable, thus e and g to be uncontrollable. The fixpoint found while starting with

$$R_0 = \langle \{f, w, c\}, (f|w|c)^* \rangle$$

is given in Figure 7. This controller leads to the same connection as the previous one, again, $P \parallel R_1$ is given by Figure 6. It is surprising that the system is controllable, event when event g is uncontrollable: the goat can jump into the boat whenever he wishes and is able to (e.g., if the farmer decides to go alone to the other side it is possible that, just before he leaves, the goat jumps into the boat). The controller in Figure 7 forbids all moves that may lead to the occurrence of event e taking into account the free moving of the goat. It is left to the reader to verify that the connection of P and this controller indeed leads to the desired behavior.

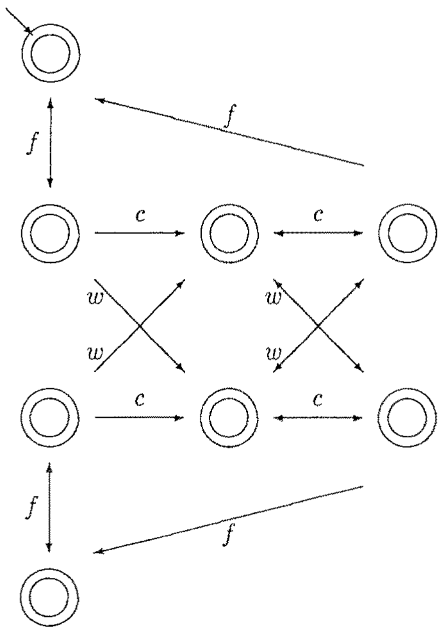

Figure 7. A second controller.

12. Conclusions

In this article we have defined discrete event systems using perhaps the simplest possible definition. In spite of its simplicity it allows to model connection of systems. Even asynchronous connection can be modeled. Also (dead)lock of one or more systems can easily be defined and useful theorems about lock-free connections can be derived. Once more we want to emphasize that the systems under consideration need not be regular. All derived properties are also valid for nonregular systems. In fact, we do not impose any restrictions on the systems. If, however, the systems are regular, we can use algorithms on state graphs to compute the lock-free controller effectively.

Starting with a controller R that is as general as possible (i.e., complete with respect to its alphabet) leads to a fixpoint that has minimal restrictions on the resulting lock-free connection. In Smedinga [1991] this idea is discussed in more detail.

The method developed for finding the greatest subsystem that leads to a lock-free connection can also be used if some controller R needs to be computed in order to have a connection $P \parallel R$ that meets certain criteria and is free of lock. Such controllers can be found by first looking only at the criteria and compute the greatest controller that is satisfactory. Second, we can compute the greatest subsystem of this controller such that the connection is free of lock. For example, the problem of computing a controller such that the behavior of the connection lies between a minimal and a maximal behavior, can be solved this way, see Smedinga [1991].

We modeled discrete event systems using triples consisting of alphabet, behavior set and task set. In this way it was possible to define the notion lock on one single system. It also gives the possibility to define unrealistic systems (GDESs). Such unrealistic systems are used in this article to model separate traces (the set **lock**(P) for example). GDESs also play a crucial role when a controller R is to be found such that the external connection $P \rceil\lceil R$ needs certain requirements. This last problem can be solved using the so called reflection operator, which returns an unrealistic system, but nevertheless, leads to the greatest desired controller R such that $L_1 \subseteq P \rceil\lceil R \subseteq L_2$; see Smedinga [1992b].

References

Hoare, C.A.R., 1985. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall.

Hopcroft, J.E., and Ullman, J.D., 1979. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley.

Kaldewaij, A., 1988. *A formalism for concurrent processes*. Ph.D. thesis, Department of mathematics and computing science, Eindhoven University of Technology.

Li, Yong, and Wonham, W.M., 1988. Deadlock issues in supervisory control of discrete event systems. *Proc. 1988 Conf. Information Science and Systems*, Princeton University, Princeton, NJ.

Milner, R., 1980. *A Calculus for Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer-Verlag.

Peterson, J.L., 1981. *Petri Net Theory and the modelling of Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Ramadge, P.J., and Wonham, W.M., 1987. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1). See also Systems Control Group report 8515, Department of Electrical Engineering, University of Toronto.

Ramadge, P.J., and Wonham, W.M., 1989. The control of discrete event systems. *Proc. IEEE*, 77(1).

Smedinga, R., 1988. Using trace theory to model discrete events. In *Discrete Event Systems: Models and Applications*, Varaiya, P. and Kurzhanski, A.B. (eds.) Lecture Notes in Control and Information Science, No. 103, Springer-Verlag. Workshop Sopron, Hungary, IIASA.

Smedinga, R., 1989. *Control of discrete events*. Ph.D. thesis, University of Groningen.

Smediniga, R., 1991. An effective way to undo a discrete event system of its (dead)lock. *Preprints Proc. IFAC Symp. Design Methods for Control Systems*, Zürich.

Smedinga, R., 1992a. Discrete event systems. Course notes, Department of Computing Science, University of Groningen.

Smedinga, R., 1992b. The reflection operator in discrete event systems. Technical report CS9201, Department of Computing Science, University of Groningen.

van de Snepscheut, J.L.A., 1985. *Trace Theory and VLSI Design*. Lecture Notes in Computer Science, vol. 200, Springer-Verlag.

Udding, J.T., 1984. *Classification and composition of delay-insensitive circuits*. Ph.D. thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.