Discrete Event Control Motivated by Layered Network Architectures

Ard Overkamp

STELLINGEN

behorende bij het proefschrift

Discrete Event Control Motivated by Layered Network Architectures

van

Ard Overkamp

1

Alleen van de liefde kun je leven.

2

Democratie in een niet homogene samenleving leidt al gauw tot een dictatuur van de meerderheid.

3

Het is voor twee OIO's of AIO's, met beiden een tweedejaars salaris, onmogelijk om via een woningbouwvereniging of makelaar legaal samen een woning te huren in Amsterdam, uitgezonderd Amsterdam Zuidoost.

4

Gezien het aantal bergen in Nederland is het verwonderlijk dat de mountainbike veel populairder is dan de ligfiets.

5

De verplichting om stellingen te produceren brengt promovendi in de verleiding om ongenuanceerde uitspraken te doen.

6

De automatische zonneschermen van het CWI zullen niet wezenlijk slechter functioneren indien de regeling gekoppeld wordt aan het space-time pendulum.

Discrete Event Control Motivated by Layered Network Architectures

Ard Overkamp

RIJKSUNIVERSITEIT GRONINGEN

Discrete Event Control Motivated by Layered Network Architectures

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van
Rector Magnificus Dr. F. van der Woude
in het openbaar te verdedigen op
vrijdag 8 november 1996
des namiddags te 1.15 uur

door

Ard Andreas Franciscus Overkamp

geboren op 16 oktober 1966
te Winterswijk

Promotor: Prof. dr. ir. J. H. van Schuppen

Vooraf

In het hierna volgende proefschrift staan de resultaten van het wetenschappelijk onderzoek waar ik me de afgelopen vier en een half jaar mee bezig heb gehouden. Vanzelfsprekend heb ik dit niet in m'n eentje gedaan, en vanzelfsprekend heb ik me niet alleen met wetenschappelijk onderzoek bezig gehouden. Iedereen die bijgedragen heeft aan de totstandkoming van dit proefschrift, en iedereen die meegeholpen heeft deze periode tot zo'n onvergetelijke tijd te maken wil ik hierbij bedanken.

Een paar mensen wil ik toch even speciaal vermelden. Ten eerste m'n promotor en begeleider Jan van Schuppen. Ondanks zijn vele verplichtingen had hij het nooit te druk om met mij over het onderzoek te discussiëren. Altijd sneller dan verwacht wist hij mijn schrijfsels te lezen en van het nodige commentaar te voorzien. Ik waardeer het ook zeer dat hij mij in de gelegenheid heeft gesteld om vele reizen naar het buitenland te maken. Onze activiteiten buiten het werk om heb ik altijd heel plezierig gevonden.

De overige collega's van de afdeling BS wil ik bedanken voor de altijd goede sfeer op het CWI. Ik hoop alleen dat voldoende mensen het belang van een degelijke koffiepauze blijven inzien.

Een speciale vermelding verdienen ook de leden van de afdeling zuiver jongleren. De gezellige woensdagavondsessies vormden een welkome onderbreking van het wetenschappelijk gegoochel tijdens de rest van de week. Ik hoop nog regelmatig in de gelegenheid te zijn om een heleboel kegels met ze over te gooien en dit daarna theoretisch uit te werken.

Mijn (ex-)teamgenoten van DVVA bedank ik voor de vele plezierige en sportieve uurtjes op en rond het voetbalveld.

Mijn familie, op wiens hulp ik altijd kan rekenen, mag in dit stuk niet

ontbreken. Ondanks hun bescheiden en nuchtere instelling weet ik dat ze de afgelopen tijd heel erg met me hebben meegeleefd.

En, Jacqueline, zonder jou zou alles niet half zo lekker gesmaakt hebben.

Amsterdam, september 1996

Ard Overkamp

Table of Contents

Chapter 1
Introduction

A discrete event system is a system whose dynamics is characterized by the occurrences of events at possible unknown irregular moments in time. For example, an event may correspond with the arrival of an elevator at a certain floor, the arrival of a message in a communication system, the failure of a machine in a manufacturing system, or the completion of a computation in a computer system. The state of a discrete event system may change abruptly at the occurrence of an event. In between two events the system remains in the same state. The behavior of a discrete event system is described by the occurrences of events.

For a large class of systems the exact time information when an event occurs is not important. For example, to make sure that a buffer does not contain more than one element, it is not necessary to know when elements are added to the buffer or removed. It is sufficient to prevent the addition of a new element before the previously added element is removed. In terms of events: allow an 'add-event' only after a 'remove-event'. The behavior of such a discrete event system is characterized by the order in which events can occur. No information is used on the time when an event can occur or on the probability that a certain event will occur. These discrete event systems are usually referred to as *logical discrete event systems.* In this thesis our attention will be restricted to these systems. In the sequel the term discrete event system is used to denote a logical discrete event system.

Discrete event systems have been investigated in the field of system and control theory since the early 1980's [50, 51, 62]. This field of research is commonly referred to by the name *Supervisory Control Theory*. An overview on the subject can be found in [45, 52, 59]. Various control problems for discrete event systems have been introduced and solved. The general objective of these control problems is to synthesize a controller (usually called a *supervisor*) that influences a given *uncontrolled system* such that the *controlled system* (the combination of uncontrolled system and supervisor) satisfies a given *specification*.

Example 1.1 Consider a telephone network. Subscribers can generate events such as 'taking the receiver off the hook', 'replacing the receiver', and 'press a button'. The telephone network itself also generates events, such as 'ring the bell', 'start the dial tone', and 'establish a connection'. Some sequences of events represent illegal behavior. For instance, a bell should not ring if the subscriber is not called. Other sequences represent legal behavior. For instance, a connection should be established if the right protocol is followed by both subscribers. The caller should have taken the receiver off the hook, waited for the dial tone, dialed the correct number, and so on. The sequences that represent the legal behavior are described by the specification. Some illegal event sequences are restricted by the hardware of the telephone network. A receiver can only be replaced after it is taken off the hook. These sequences are modeled by the uncontrolled system. The supervisor can influence the behavior of the uncontrolled system by disabling a set of events after the observation of each event. Some events cannot be disabled. For instance, the system cannot prevent that the subscriber replaces the hook on the receiver. The event corresponding with this action cannot be disabled by the supervisor. Events that cannot be disabled are called *uncontrollable*.

The task of supervisory control is to synthesize a supervisor such that only legal sequences will be generated.

Of course, Complete telephone networks are far too complex for supervisory synthesis. Also the performance measures (using time and probability) cannot be neglected in these systems. The example is intended to illustrate the basic concepts of supervisory control. These concepts apply also to other supervisory control problems.

Algorithms have been derived that can automatically synthesize a supervisor which solves the given supervisory control problem. Practical problems for which supervisory control theory has been applied include the transaction execution in database systems [34] and the design of a rapid thermal multiprocessor [4].

In the field of computer science many systems can be regarded as discrete event systems. In this thesis the existing methods from supervisory control theory are adapted and extended such that discrete event control problems from computer science can be solved. In the next section some general characteristics of discrete event systems from computer science are discussed. The supervisory control framework that is presented in this thesis is motivated by these characteristics.

1.1. Discrete Event Systems in Computer Science

One of the main differences between conventional continuous variable systems and discrete event systems is linearity. Continuous variable systems can be described, or approximated, by linear equations. Linearity makes results scalable. The analyses of a chemical reaction in a tank with 5000 liter is not more difficult then the same reaction in a test-tube. Discrete event systems lack a similar property such as linearity. Yet, discrete event systems are often large and complex systems. Consider for example internet, the complete financial administration of an international bank, or a communication link with a satellite. How did computer scientist and engineers deal with these large and complex systems? Somehow they must have found a way to handle these systems. After all, the examples given above are of existing discrete event systems.

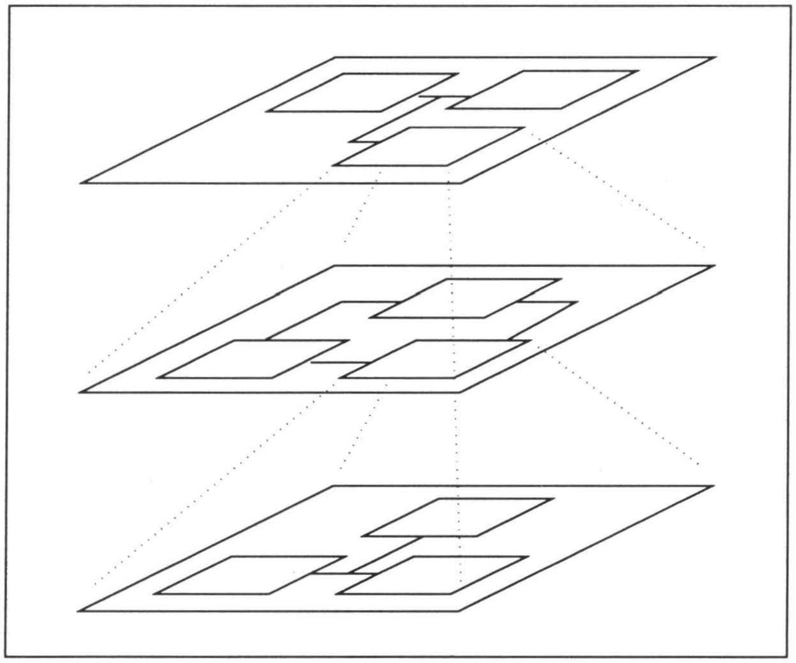


Figure 1.1: Layered structure of discrete event systems.

The main tool for handling large and complex systems is *abstraction*. With abstraction one stresses certain details by disregarding the other details. An example of an abstraction is a function or procedure in a programming language. The heading of a function describes which arguments are needed and what the type of the result will be. In addition it is a good policy to add a few lines of comments which describe what the function does. This information is sufficient to use the function in the rest of the program. The function heading stresses what the function does and how it should be called. It hides the actual implementation (the body of the function) from the rest of the program.

Usually in computer science, systems are designed top down. First, on the highest level, the system is modeled using very abstract terms. For instance, the highest level could consist of the three parts, 'get input', 'do calculations', and 'produce output'. Next, each of the parts, or modules, is refined into a more concrete description. The 'get input' part could be modeled by the modules 'show general info', 'get personal identification', 'repeat get query until stop'. Each of these parts can be refined further until an executable program is obtained. With this approach the whole system is structered into a hierarchy of layers. The most abstract layer on top and more concrete implementations of each module in the layers below. See Figure 1.1. Using a hierarchical structure the design of a large and complex system can be decomposed into smaller and simpler subproblems. In each layer an implementation should be designed for the abstract description of the module given in the layer above. The design may contain elements that will be made more concrete in the layers below.

Supervisory control of hierarchical discrete event systems has been explored by H. Zhong, K. C. Wong, and W. M. Wonham [60, 61, 64, 65]. Their objective is to perform the synthesis process on a higher level of abstractions. They have formulated conditions under which the synthesized higher level supervisor will have a lower level implementation. Our aim is not to perform the synthesis process on a higher level, but to find a supervisor such that the implementation can be used at a higher level. This poses some restrictions on the control framework that have not been considered before in the field of supervisory control. These restrictions are discussed in Section 2.5.

In each layer each module of the layer above is decomposed into a number of smaller modules. In the lowest layer this will result in a large number of small modules that form together the implementation of the system. In Section 2.5 it is shown that our control framework is well suited for control approaches that make use of the modular structure of discrete event systems.

Concluding, a typical discrete event system used in computer science is a large and complex system, it lacks a property such as linearity, it is structured into layers, and it consists of a large number of small modules.

1.2. Design of Discrete Event Systems

In the previous section some characteristics of discrete event systems from computer science are described. Up till now these systems are designed manually. In this section a brief introduction will be given to concepts that are used to design these systems. We will discuss the concepts of simulation, verification, and formal specifications. Supervisory control theory deals with the automatic synthesis of solutions for discrete event design problems. In the next section it will be described how automatic synthesis relates to the discussed design concepts. A small software programming problem will be used as illustration. Consider the following problem specification.

> Write a program that gets n and prints the numbers 1 to n in reversed order.

Given this specification the following implementation has been written.

```
PROGRAM reverse1
    get $n$
    WHILE $n \neq 0$ DO
        print $n$
        $n := n - 1$
    ENDWHILE
END
```

The question is now whether this program is a solution to the given problem. Or, in other words, whether this implementation implements the given specification. The easiest and most frequently used way to test the program is to do a *simulation*. The program is executed with different inputs. Afterwards it is checked whether the output is according to the specification. A simulation with our example program gave the following result.

n	output												
5	5	4	3	2	1								
13	13	12	11	10	9	8	7	6	5	4	3	2	1
-5	-5	-6	-7	-8	-9	...							

Obviously the reaction to $n = -5$ is not what it should be. But, what is the proper output to negative numbers? The specification does not make an explicit statement about it. It states that the sequence 1 to n should be printed in reverse order. But what is the reverse order of the sequence 1 to -5? Is it $-5\ -4\ -3\ -2\ -1\ 0\ 1$, or is it $1\ 0,\ -1\ -2\ -3\ -4\ -5$? Or can we assume that negative numbers do not have to be considered? The specification does not give an answer to these questions. It is ambiguous. It can be interpreted in different ways. In many situations ambiguous specifications are unacceptable. Therefore, in these situations *formal specifications* are used. A formal specification is an unambiguous mathematical description of the required behavior for the implementation. The specification that the numbers 1 to n should be printed in reversed order could be formalized by the following code.

$\mathrm{P} = \text{get } n\ ;\ \mathrm{P}(n)$

```
P($\not{n}$)($n < 1$) : P
      ($n \geq 1$) : print $n$; P(n-1)
```

We will not go deeper into the syntax and semantics of the used language as it will not be used further on.

The first remark one can make on formal specifications is that it must be learned by the users. Unlike English, one cannot assume that a programmer at the other side of the world will understand this specification. This is a drawback of formal specifications.

But formal specifications have a lot of advantages. We already mentioned that they are unambiguous. This is especially useful if an implementation is designed by a different manufacturer. The formal specification serves as an independent reference (free from subjective interpretations) to decide whether a product (implementation) is as ordered (as specified).

Formal specifications can be used to test in an early phase certain properties of the system. In this way misconceptions can be detected early and the correction will be relatively cheap. A conceptional error that is detected during the simulation phase will be much more expensive to solve.

Formal specifications can be used to automatically generate test sequences for the simulation. The objective is to generate sequences that will excite the system as much as possible. Especially, sequences must be generated that will test exception cases and border situations (buffer overflow, unexpected inputs, etc.).

A formal specification can be used to guide the design of the implementation. In subsequent steps the more abstract terms in the specification can be refined into more concrete implementations. After each step the implemented part can be tested for correctness. In each step the system changes only little. The test will therefore be relatively easy.

And last but not least, formal specifications are a prerequisite for automatic verification and automatic synthesis that will be discussed later.

Normally a specification is given on a higher level of abstraction than the implementation. The implementation describes details that are not used in the specification. The specification describes *what* a system should do. The implementation describes *how* this is achieved. For the sake of simplicity the implementation is formulated on the same level of abstraction as the specification in this example.

The program has been adapted according to the formal specification.

```
PROGRAM reverse2
    get $n$
    IF $n > 1$ THEN
        WHILE $n \neq 0$ DO
            print $n$
            $n := n - 1$
        ENDWHILE
    ENDIF
END
```

A simulation with this program gave the following result.

n	output												
5	5	4	3	2	1								
13	13	12	11	10	9	8	7	6	5	4	3	2	1
-5													

And a couple of hundred other inputs have been tested that all gave the expected output. However, the program is not correct. The program does not give any output on $n = 1$. Yet, according to the specification a 1 should be printed. This example illustrates that simulation can never guarantee correctness. It is impossible to test all possible inputs. There is always the risk that an error occurs on an input that has not been tested.

Often, it is absolutely necessary to guarantee the correctness of a software program. In these cases a formal proof that the program satisfies the specification is required. This is called *verification* [5, 33].

Usually the verification of a system is very difficult. Attempts have been made to automate the verification process [12, 33]. However, it can be shown that no automatic verification procedure will work on all programs. For some programs the verification procedure will keep on running without ever giving the answer whether the program is correct or not. The problem is said to be *undecidable*. Typically, verification procedures have to check all possible states of the program. If the state space is infinite, it will require infinite time to check all states. In other words, the verification procedure will never stop. But even if the state space is finite it may be so large that automatic verification is unfeasible. In fact, most practical applications have either an infinite state space or a state space that is too large to be checked. Recall that systems typically consist of a large number of small modules. It can be shown that the state space size of the complete system can be approximated by the product of the state space sizes of all the modules. So, if there are n modules, each with about k states then the complete system will have approximately k^n states. The size of the complete system is exponentially related to the number of modules. Therefore, also the complexity of the verification procedure, which checks every state at least once, will be exponentially related to the number of modules in the system. Because of this exponential relationship, automatic verification will be unfeasible even for systems with a moderate number of modules. This phenomena is called the *state space explosion problem*.

Because of these difficulties there are still people who believe that verification will never be useful for real-life applications. They have more confidence in modeling methodologies that help designers to write good programs. In addition to that they try to find better simulation methods that will give as much confidence in the correctness of programs as possible.

However, a growing group realizes that verification has its advantages. Especially for the design of hardware has verification shown its usefulness. Currently companies are emerging that make their living with formal verification.

Verification *guarantees* the correctness of a program. It is expected that in the future guaranteed correctness of software will be more important. Consider the situation in which the software vendor will be held liable for the consequences of software errors. The risk of a financial disaster might compensate the costs of verification.

The question is not whether verification is useful at all, but for what applications is it worthwhile.

Currently numerous results are being published on methods to reduce the complexity of verification procedures. See for instance [3] as a representative reference. Important within this research is to use the modular structure of discrete event systems.

1.3. Synthesis of Discrete Event Systems

In the previous section some concepts on the design of discrete event systems are discussed. In this section the concept of automatic synthesis is treated.

Consider the design methods as discussed in the previous section. First a formal specification must be written. This is done manually. Next an implementation is created. Also manually. This implementation is tested (manually, semi-automatic, or automatic). If the implementation contains an error then it needs to be redesigned. Again manually. All this manual work costs a lot of effort and money. The idea of synthesis is to automate the creation and the test phase. Taking into account the inherent difficulties of automatic verification one can ask the question what can be expected from automatic synthesis. This question will be treated in this section. Consider the following naive approach to automatic synthesis.

```
REPEAT
    choose implementation
    test implementation
UNTIL no errors are found
```

As a computer does not have creative powers, it must choose an implementation. It can be expected that the chosen implementation will not be immediately correct. So the repeat-until loop will be traversed several times. As in each loop the implementation is verified, it is expected that automatic synthesis has a much worse complexity than automatic verification. However, we will see that the choice part and the test part can be closely intertwined. This will result in a synthesis method that, for a large class of problems, has a complexity comparable to that of verification.

In the previous section it was stated that for most real-life problems automatic verification is too complex. But, we also mentioned that it is still worthwhile to explore the possibilities of verification techniques, because verification can guarantee the correctness of a program. The same holds for automatic synthesis. It is still too complex to handle real-life problems, but it guarantees the correctness of the implementation. The advantage of automatic synthesis over verification is that the implementation does not have to be designed by hand.

As the synthesis method that is presented in this thesis is closely related to verification methods, it is expected that complexity reducing algorithms for verification can be adapted to the synthesis procedure. Further research is needed in this direction.

In this thesis automatic synthesis will be considered from a supervisory control theory point of view. The objective of supervisory control is to generate a supervisor such that the controlled system satisfies a given specification. Automatic synthesis can be considered as a supervisory control problem. The uncontrolled system can be regarded as an incomplete implementation. The task is then to find the missing part of the implementation, given the incomplete implementation and the specification.

1.4. Layered Network Architectures

Computer networks are essential for the functioning of today's society. More and more services are offered that make use of computer networks. All these services need to be designed and tested. Moreover, the correctness of these service becomes increasingly important. Think for instance of distributed databases with confidential information, money transfers, or the financial administrations of large international companies. This makes computer networks an interesting field for supervisory control. Although, our supervisory control framework is motivated by computer networks, the results are certainly not limited to this field.

Computer networks are very complex systems. Similar to the abstraction method described in Section 1.1 these systems are organized in a layered structure [57]. The purpose of each layer is to offer a certain *service*. For instance, one layer makes sure that messages are transported to the correct node. Another layer guarantees that no messages will get lost. Each layer uses the service offered by the layer below and hides its implementation from the layers above. To achieve a service, the layer at each node sends control messages to the same layer at the other nodes. The rules for these conversations of control messages are called the *protocol* of that layer. In Figure 1.2 the layered structure of the standard reference model for Open Systems Interaction (OSI) from the International Standards Organization (ISO) is shown [57, 66]. The protocols are indicated by dotted lines.

The control messages are not sent directly from layer n at one node to the same layer at another node. The messages are transferred to the layer immediate below. Depending on the protocol of that layer the message will

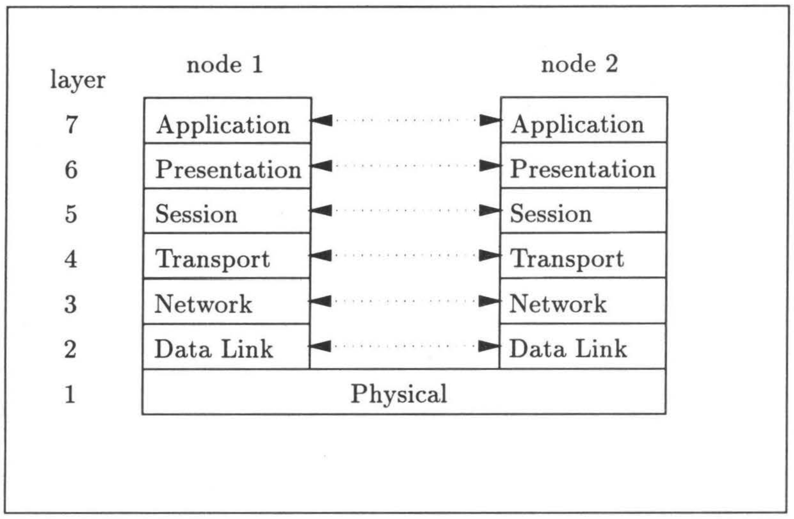


Figure 1.2: The ISO-OSI reference model.

be transferred further down, possibly with some extra control information attached. This continues until the lowest layer is reached. There the message is transported to the other node where it is propagated upward. On its way up the extra control information is removed and the message is finally delivered to layer n of this node. The actual implementation of all the layers below layer n is hidden from layer n. To this layer it appears as if layer $n-1$ transported the message to the other node according to its service. The service of layer $n-1$ is the only information that layer n has of the underlying message transport mechanisms.

The protocol design problem for layer n is to find a protocol that achieves the layer n service using only the service offered by layer $n-1$. The similarity with the supervisory control problem is evident. Consider the service of layer $n-1$ as uncontrolled system, the protocol of layer n as supervisor, and the service of layer n as specification. Then the protocol design problem can be regarded as a supervisory control problem.

1.5. Discrete Event Control for Layered Network Architectures

The layered network architecture places constraints on the supervisory control problem. Some of these constraints have not been considered before in the literature on discrete event systems. One constraint which has been considered before is caused by the decentralized nature of networks. Communication protocols do not consist of one monolithic part that resides at one node. They consist of several independent parts, one at each node. Each part observes the messages received at that node, and each part can transmit messages from that node. They do not have any information on the messages that are received and transmitted at the other nodes. This aspect of the protocol design problem corresponds to the decentralized supervisory control problem [13, 30, 53, 54, 55]. In this control problem a set of supervisors must be synthesized, each observing only part of the system and each capable of influencing only part of the system. The combination of all supervisors and the uncontrolled system must be an implementation of the specification. The literature on this topic shows that the decentralized control problem is very hard. Only limited results have been presented so far. In Chapter 6 the decentralized supervisory control problem will be discussed. Decentralized control problems are strongly related to control problems of game and team theory. In these fields the notion of a Nash equilibrium plays an important role. It will be shown that Nash equilibria are also important for decentralized supervisory control theory. It will be shown that a stronger version of the Nash equilibrium condition characterizes all maximal solutions. This result holds irrespective of the event sets which the supervisors can observe. If these event sets are disjoint, then a weaker condition can be given. In this case, a solution is maximal if and only if an equivalent canonical solution forms a Nash equilibrium. These results provide new insight in the fundamental properties of decentralized control problems. More research is necessary to extend the results to effective synthesis methods.

The results of Chapter 6 have been published as an extended abstract in the proceedings of WODES 96 [46]. A longer version, including all the proofs, is in preparation.

The decentralized nature of the protocol design problem can be regarded as the horizontal dimension of the problem. The vertical dimension corresponds to the layered structure of the problem. Results in this direction are more promising. The two dimensions of the protocol design problem will be treated independently. In Chapter 6 the decentralized control problem is discussed. In the Chapters 2 till 5 control problems related to the layered structure are treated. In the rest of this section some characteristics caused by the layered structure of the protocol design problem will be discussed.

The objective of supervisory control is to find a supervisor such that the controlled system satisfies the specification. But, when does an implementation satisfy a specification? The answer to this question comes from the layered structure. The service of layer n is used by the protocol of layer $n+1$. When an implementation for the service of layer n (i.e. the protocol of layer n) is designed, then it is not yet known what the protocol for layer $n+1$ will be. However, the implementation must be such that it works properly together with this protocol. With 'working properly' we mean that the system does not generate any illegal events, and that it does not deadlock. The implementation must be such that it works properly with any layer $n+1$ protocol that works properly with the layer n service. This question has not been addressed in the literature on supervisory control up till now. It will be shown in Chapter 2 that the implementation relation used in this thesis satisfies this requirement.

In order to achieve a certain service, the protocol uses extra implementation events (e.g. events corresponding with control messages). These implementation events are not used in the specification. The specification describes *what* service the layer offers. It does not describe *how* this service is, or should be, implemented. The specification uses only part of the events that are used in the implementation. Therefore, the specification is called a *partial specification*. The supervisory control problem with partial specification is discussed in Chapter 4. It will be shown how, and under what restricitons, this control problem can be reduced to the basic supervisory control problem. The results of this chapter will appear in [44]. In [43] the results are presented without proofs.

1.6. Nondeterminism

A discrete event system is called *deterministic* if at each state, each event uniquely determines in which state the system will be after the execution of that event. See Section 2.1 for a formal definition of a deterministic discrete event system. Traditionally, supervisory control theory only deals with deterministic systems. In computer science it is more common to use nondeterministic systems. A system is nondeterministic if there is a state from which multiple states can be reached by identical events.

Nondeterminism reflects the lack, or the deliberate hiding, of information. Reality may be considered deterministic, but models are an abstraction of reality. Models stress some aspects of reality by hiding irrelevant details. Hiding of details causes systems to exhibit nondeterministic behavior. Of course, models can be made deterministic by including also the unimportant details and stating which details are important and which are not. However, this will lead to unnecessarily complex models.

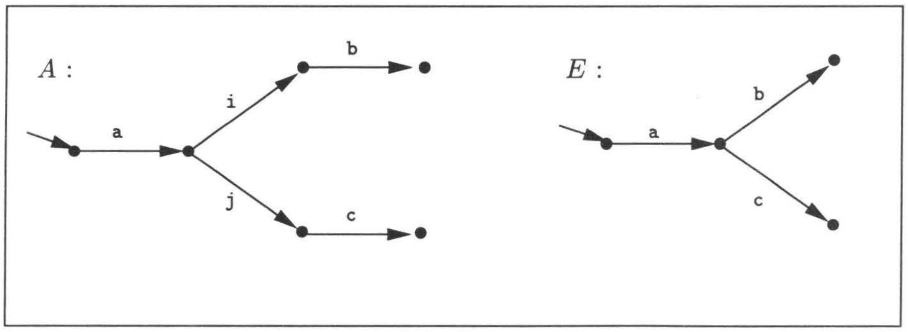


Figure 1.3: Does system A implement partial specification E?.

Partial specifications also lead to issues concerning nondeterminism. Consider the systems A and E shown in Figure 1.3. System E can first execute event a and then either event b or event c. See Section 2.1 for a more formal introduction of the used representation. Let E be a partial specification and let A be a candidate implementation. After event a system A can either execute implementation event i and then event b, or implementation event j and then event c. The question is whether system A implements partial specification E?

In the context of layered architectures, E corresponds with the service of a certain layer n. This service is used by the protocol of layer $n+1$. According to service specification E the protocol of layer $n+1$ expects that after event a event b can be executed. However, when system A is used instead of system E then event b may not be executable after event a. System A may execute implementation event j after which only event c can be executed. This may lead to a deadlock if system A is combined with the protocol of layer $n+1$. System A does not implement partial specification E.

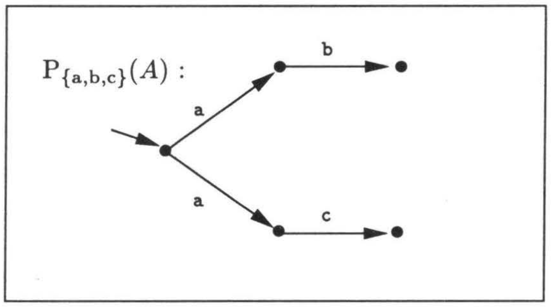


Figure 1.4: The projection of system A on event set $\{a, b, c\}$.

If system A is considered without the implementation events (system A is *projected* onto event set $\{a, b, c\}$), then it exhibits nondeterministic behavior. In Figure 1.4 the projection of system A onto event set $\{a, b, c\}$ is shown. After event a it can be either in a state in which it can only execute event b, or in a state in which it can only execute event c. System A does not implement E because, when projected, it is more nondeterministic than system E. This example illustrates that partial specifications lead to questions concerning the nondeterministic properties of systems. A framework for supervisory control problems with partial specification must be able to handle these nondeterministic aspects.

1.7. Discrete Event Control for Nondeterministic Systems

In Chapter 2 a supervisory control framework is presented in which nondeterminism is fully integrated. The uncontrolled system, the specification, and the supervisor can all be nondeterministic. In Section 2.5 the framework is compared to other frameworks. In Chapter 3 it is shown how supervisory control problems can be solved in this framework.

Results on this framework have been published in [42, 43, 44]. In [44] the comparison of frameworks from Section 2.5 has also been included. The synthesis method used in that paper is based on languages. It is the same method as used in Section 3.4 for the direct computation of the deterministic least restrictive solution. A paper including the synthesis method used in this thesis will be submitted to a journal.

In the literature several formalisms are proposed to model discrete event systems. Formalism have been proposed based on Petri nets [1, 9, 25, 40, 47], automata [26, 33, 52], and process algebra's [2, 23, 28, 39]. The purpose of this thesis is to explore fundamental properties of discrete event systems. These fundamental properties hold irrespective of the used modeling formalism. We will use automata [26] to represent discrete event systems. Automata are based on the concept of 'state'. This is a key concept when reasoning about the behavior of discrete event systems. We are confident that the results in this thesis also hold for other modeling formalisms.

Two models are considered equivalent if their corresponding systems have the same behavior. The behavior of a system corresponding with a model is described by the *semantics* of the modeling formalism. Deterministic discrete event systems are adequately described by the sequences of events they can generate. A sequence of events is called a *trace*. The set of all traces that a system can generate is called the *language* of the system. The corresponding semantics is called *language semantics* or *trace semantics*. In general also the concept of *completed trace* is used. A completed trace indicates that a certain task is completed. A semantics that uses completed traces is called a *completed trace semantics*. Typically, supervisory control frameworks for deterministic discrete event systems are based on completed trace semantics. It will be shown in Section 2.5 that in these frameworks the proper behavior of an implementation inside an arbitrary environment cannot be guaranteed.

Language semantics and completed trace semantics are not capable of describing the nondeterministic properties of discrete event systems. Therefore, *failure semantics* will be used in this thesis. Failure semantics has been introduced by C. A. R. Hoare et al. [10, 11, 23] as a mathematical foundation for the programming language CSP. Other semantics are known that can handle nondeterminism [2, 39]. Yet, failure semantics is the weakest. In supervisory control theory one hopes to find a necessary and sufficient condition for the existence of a supervisor. The weakest possible semantics is required for such a condition. The combination of Theorem 2.17 and Theorem 3.2 provides a necessary and sufficient condition for the existence of a supervisor such that the controlled system behaves properly in any environment.

Nondeterminism is used by K. Inan [27] to represent the class of all solutions in a single finite state automaton. However, the framework he uses is based on deterministic systems and completed trace semantics. The question whether an implementation satisfies a specification depends only on the completed traces that the systems can generate.

The main difference between the approach presented in this thesis and the approach presented by R. Kumar and M. A. Shayman [32, 56] is that the latter explicitly states whether events are generated by the uncontrolled system or by the supervisor. This approach requires a stronger semantics than failure semantics. They use the so called *trajectory semantics* for their models. Originally, Ramadge and Wonham illustrated the connection between uncontrolled system and supervisor by assuming that the uncontrolled system generates all events. However, the underlying theory does not rely on this assumption. The systems describe which sequences of events can be generated, independent from the question where these events are generated. It is a strong point of this theory that the location of event generation is unimportant. Similar, the framework presented in this paper does not depend on any assumption concerning the place where events are generated.

Kumar and Shayman use a language as specification, whereas in this thesis a (nondeterministic) process is used. In Section 2.5 the advantages of a nondeterministic specification are discussed.

Other control frameworks for nondeterministic systems have been presented in the literature. M. D. DiBenedetto, A. Saldanha, and A. Sangiovanni-Vincentelli have presented a framework based on I/O automata [16, 17]. The approach of K. G. Larsen and others is based on bisimulation semantics [29, 35].

In Chapter 4 the supervisory control problem with partial specifications is treated. It is shown how, and under what restrictions, this control problem can be reduced to the basic supervisory control problem. Inan also discusses the control problem with partial specifications [27]. However, as mentioned before, he uses completed trace semantics. It is not guaranteed that the solutions proposed by Inan will behave properly inside an arbitrary environment.

A supervisor cannot always observe all events. For example, a controller at one node of a network can observe only those events that correspond with messages received or transmitted at that node. It is said that the supervisor has only partial observation [13, 36]. The supervisory control problem with partial observation is treated in Chapter 5.

A supervisor influences the uncontrolled system by disabling certain events. The controlled system cannot execute events that are disabled by the supervisor. The set of events that a supervisor can disable is called the set of *controllable* events. Results from the literature show that the supervisory control problem with partial observation is much harder if the supervisor can disable events that it can not observe [13, 36]. In Section 5.1 the supervisory control problem with partial observation such that the supervisor can observe what it can control is discussed. This control problem has a straightforward solution. The result has been published in [43]. In Section 5.2 the situation in which the supervisor can control events that it cannot observe is treated. It will be shown that any supervisory control problem can be modeled in such a way that all controllable events are observable. This result shows new and important concepts underlying the modeling choices of supervisory control problems. A paper containing these results will be submitted to a journal.

Chapter 2
Basic Supervisory Control Problem

Discrete event systems are systems that are characterized by the sequences of events that they can accept or execute. From an abstract point of view it is not important whether the system actually generates events or whether it restricts the possible events that are generated somewhere else. The important point is that the behavior of the system can be described by sequences of events.

Control of discrete event systems was first introduced by P. J. G. Ramadge and W. M. Wonham. See [45, 52, 59] for an overview on the subject. In the framework proposed by Ramadge and Wonham, a system is described by the sequences of events that it can generate, i.e. the language. Also the sequences that represent completed tasks are taken into account. This framework can handle deterministic systems only. Another drawback is that systems are considered on their own. But, as was discussed in the introduction, systems have to be considered in combination with their environment. It will be shown that a framework based on complete sequences is not suited for considering systems inside an environment. In this thesis a framework will be presented that guarantees the correct behavior of implementations in any environment. The framework is based on failure semantics [10, 11, 23]. Failure semantics provides a theoretical foundation to reason about the behavior of nondeterministic discrete event systems.

A discussion on the motivation of this framework will be given in Section 2.4. It will be shown that the supervisory control framework based on failure semantics is a flexible and elegant method. It guarantees deadlock free behavior under all circumstances, it allows for powerful specifications, it forms a sound basis for modular control, and it can handle nondeterminism without extra effort.

2.1. Languages and Automata

In the literature several models are proposed to describe the behavior of discrete event systems. To name a few: languages, automata, transition systems, Petri nets, boolean expressions, process algebras, etc. In this thesis we will use representations based on languages, automata, and failure semantics. Failure semantics will be discussed in the next section. In this section formal definitions concerning languages and automata are given. Also some well known results from the literature are recalled. See [26] for a more detailed introduction to languages and automata.

Definition 2.1 Let Σ denote the finite set of all possible events or event labels. A set of events is usually called an *alphabet*. A *trace* or *string* is a finite sequence of events, $\sigma_1\sigma_2\ldots\sigma_n$ with $\sigma_i \in \Sigma$ for all $i \in 1\ldots n$. The *length* of a trace is the number of events in the trace. Let ε be the empty trace, i.e. the sequence of events with length 0. Note that $\varepsilon \notin \Sigma$. Σ^n denotes the set of traces with length n. Let $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^n$. It denotes the set of all finite traces with events in Σ. Let $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^n = \Sigma^* - \{\varepsilon\}$. A *language* is a set of traces, i.e. a subset of Σ^*. The language of discrete event system A is denoted by $\mathrm{L}(A)$.

An *automaton* is a representation of a discrete event system based on states and transitions between states. In the literature also the terms labeled transition system, finite state machine, and generator are used. The differences between these models are relatively small. In this thesis we will use the term automaton to denote all these forms. When necessary we will state explicitly which form is used.

Definition 2.2 An *automaton* in its basic form is a four tuple[1]

$$X \;=\; (\Sigma(X),\, Q(X),\, \delta(X),\, Q_0(X)),$$

where

$$\begin{array}{lcll}
\Sigma(X) & \subseteq & \Sigma & \text{is the set of events,} \\
Q(X) & & & \text{is the set of states,} \\
\delta(X) & : & Q(X) \times \Sigma(X) \to 2^{Q(X)} & \text{is the transition function,} \\
Q_0(X) & \subseteq & Q(X) & \text{is the set of initial states.}
\end{array}$$

[1]In the classical automata theory [26] a fifth element describing the marked states is included. These states are needed to represent completed traces. As completed traces are not used in this thesis this fifth element is not included in the basic form of an automaton.

When needed, extra elements will be added to this basic form. In the sequel the notation $\delta(X,q,\sigma)$ will be used instead of $\delta(X)(q,\sigma)$.

Figure 2.1 shows a graphical representation of an automaton. The nodes of the graph correspond with the states of the automaton. Let q, $q' \in Q(X)$ and $\sigma \in \Sigma(X)$. Then $q' \in \delta(X,q,\sigma)$ if and only if there exists an arrow from node q to node q' labeled σ. Small arrows that do not start at a node point to the initial states.

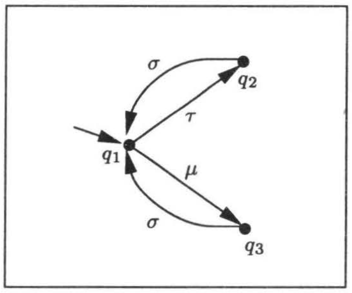


Figure 2.1: Graphical representation of an automaton.

Automata can be seen as machines that generate sequences of events. Initially an automaton is in one of its initial states. From state q automaton X can make a transition to state q' while generating event σ if $q' \in \delta(X,q,\sigma)$. If $\delta(X,q,\sigma) = \emptyset$ then X cannot make a transition from state q labeled σ. The set of traces that automaton X can generate in this way is called the *language of automaton* X, and denoted $\mathrm{L}(X)$. The term 'language of an automaton X' can be regarded as short for 'the language of a discrete event system represented by automaton X'.

The transition function can be extended naturally to traces, languages, and sets of states. Let $q \in Q(X)$, $\mathcal{Q} \subseteq Q(X)$, $\sigma \in \Sigma(X)$, $s \in \Sigma(X)^*$, and $K \subseteq \Sigma(X)^*$. Define

$$\begin{aligned}
\delta(X,\mathcal{Q},\varepsilon) &= \mathcal{Q}, \\
\delta(X,\mathcal{Q},\sigma) &= \bigcup_{q' \in \mathcal{Q}} \delta(X,q',\sigma), \\
\delta(X,\mathcal{Q},s\sigma) &= \delta(X,\delta(X,\mathcal{Q},s),\sigma), \\
\delta(X,\mathcal{Q},K) &= \bigcup_{s \in K} \delta(X,\mathcal{Q},s).
\end{aligned}$$

In the sequel we will deliberately confuse an element with the singleton set containing the element, whenever the meaning is clear from the context. For example, if the result of the transition function is a singleton set then we will write $q = \delta(X,\mathcal{Q},s)$ instead of $\{q\} = \delta(X,\mathcal{Q},s)$. Similarly define $\delta(X,q,s) = \delta(X,\{q\},s)$ and $\delta(X,q,K) = \delta(X,\{q\},K)$.

For notational convenience we will often omit the initial states as argument of the transition function.

$$\delta(X, s) = \delta(X, Q_0(X), s).$$

The language of automaton X is formally defined by

$$\mathrm{L}(X) \quad = \quad \{s \in \Sigma(X)^* : \delta(X, s) \neq \emptyset\}.$$

The definition of automaton given above is sometimes in the literature referred to as *nondeterministic automaton*. These automata are called nondeterministic because from the observation of a generated trace it cannot be uniquely determined which state the automaton has reached after execution of this trace. An automaton, X, is called *deterministic* if the initial state set is a singleton set, and if at each state $q \in Q(X)$, and for each event $\sigma \in \Sigma(X)$, $\delta(X, q, \sigma)$ contains at most one element. After observation of a trace $s \in \mathrm{L}(X)$, with X a deterministic automaton, it is always clear which state X has reached. Namely, the only element of the singleton set $\delta(X, s)$. (So, we will write $q = \delta(X, s)$ instead of $\{q\} = \delta(X, s)$.)

In the rest of this section we will discuss some properties of traces, languages and automata.

Definition 2.3 Let $s, t \in \Sigma^*$ such that $s = \sigma_1\sigma_2 \ldots \sigma_n$ and $t = \tau_1\tau_2 \ldots \tau_m$. The *concatenation* of s and t is the trace

$$st \quad = \quad \sigma_1\sigma_2 \ldots \sigma_n\tau_1\tau_2 \ldots \tau_m.$$

The concatenation of two languages K, $L \subseteq \Sigma^*$ is the language

$$KL \quad = \quad \{st : s \in K \wedge t \in L\}.$$

The *prefixes* of trace $s \in \Sigma^*$, denoted $\overline{s}$, are all traces that can be extended to s.

$$\overline{s} \quad = \quad \{v \in \Sigma^* : \exists t \in \Sigma^* \text{ s.t. } s = vt\}.$$

The *prefix closure* of a language $K \subseteq \Sigma^*$ is the set of all prefixes of traces in K.

$$\overline{K} \quad = \quad \bigcup_{s \in K} \overline{s}.$$

A language is called *prefix closed* if it is equal to its prefix closure, i.e. $K = \overline{K}$. The *choice* between languages K, $L \subseteq \Sigma^*$ is the language

$$K + L \quad = \quad K \cup L.$$

The *repetitive closure* of language $K \subseteq \Sigma^*$ is the language

$$K^* \quad = \quad \bigcup_{n \in \mathbb{N}} \{s_1 \ldots s_n : s_1, \ldots, s_n \in K\}.$$

In the sequel we will deliberately confuse a trace $s \in \Sigma^*$ with the language $\{s\}$. So an expression of the from $\sigma(\tau + \mu)^*$ denotes the language consisting of all traces of the form

$$\sigma,\ \sigma\tau,\ \sigma\mu,\ \sigma\tau\tau,\ \sigma\tau\mu,\ \sigma\mu\tau,\ \ldots$$

These kind of expressions are called *regular expressions* [26].

The *next event function* λ gives all events that are possible after a string.

$$\lambda(K, s) \quad = \quad \{\sigma \in \Sigma : s\sigma \in K\}.$$

The *ρ-function* is the complement of the next event function. It gives all events that cannot be executed after a string.

$$\begin{aligned} \rho(K, s) \quad &= \quad \Sigma - \lambda(K, s) \\ &= \quad \{\sigma \in \Sigma : s\sigma \notin K\}. \end{aligned}$$

The *language after trace $s \in K$* is defined to be

$$K/s \quad = \quad \{v \in \Sigma^* : sv \in K\}.$$

Two traces s, $s' \in K$, are called *Nerode equivalent* if the languages after s and s' are equal.

$$s \equiv_K s' \quad \Longleftrightarrow \quad K/s = K/s'.$$

Let $[s]_K$ denote the equivalence set induced by the Nerode equivalence relation, containing trace s.

$$[s]_K \quad = \quad \{s' \in K : K/s = K/s'\}.$$

If an automaton can generate trace $s \in \Sigma^*$ then it is clear that it can also generate all its prefixes. So, a language generated by an automaton is always prefix closed. Given any nonempty prefix closed language, there exists an automaton that generates this language. However, the automaton may have an infinite state space. Let the *canonical automaton representation* of the nonempty prefix closed language K be the deterministic automaton defined by

$$(\Sigma,\ Q(K),\ \delta(K),\ Q_0(K)),$$

where

$$\begin{aligned} Q(K) \quad &= \quad \{[s]_K : s \in K\}, \\ Q_0(K) \quad &= \quad [\varepsilon]_K, \end{aligned}$$

and for all $[s]_K \in Q(K)$

$$\delta(K, [s]_K, \sigma) = \begin{cases} [s\sigma]_K & \text{if } \sigma \in \lambda(K, s), \\ \emptyset, & \text{otherwise.} \end{cases}$$

Language K is called *regular* if $Q(K)$ is finite.

For any language $K \subseteq \Sigma^*$ there exists a deterministic automaton that generates K. So also for all languages generated by nondeterministic automata. In other words, for any nondeterministic automaton there exists a deterministic automaton that generates the same language. Let X be a (nondeterministic) automaton. Define

$$\mathrm{Det}(X) = (\Sigma(X),\ 2^{Q(X)},\ \delta(\mathrm{Det}(X)),\ \{Q_0(X)\}),$$

where

$$\delta(\mathrm{Det}(X), \mathcal{Q}, \sigma) = \{\delta(X, \mathcal{Q}, \sigma)\} \qquad \text{for all } \mathcal{Q} \in 2^{Q(X)} \text{ and } \sigma \in \Sigma(X).$$

The state space of $\mathrm{Det}(X)$ is the power set of $Q(X)$. It contains as states the subsets of $Q(X)$. The size of $Q(\mathrm{Det}(X))$ is $2^{|Q(X)|}$, i.e. exponentially in the size of the state space of Q. Worst case, every subset of $Q(X)$ is needed to represent $\mathrm{Det}(X)$. Therefore, the conversion is said to have a worst case exponential complexity. However, in practice only those subsets are needed that are reachable by a trace of $\mathrm{L}(X)$. So this complexity is only worst case [26].

As $\mathrm{Det}(X)$ is deterministic the result of $\delta(\mathrm{Det}(X), \mathcal{Q}, \sigma)$ is a singleton set. It contains as single element the set $\delta(X, \mathcal{Q}, \sigma)$. Likewise, the set of initial states contains one element: $Q_0(X)$.

2.2. Processes

Traditionally in discrete event control only deterministic systems are considered. Two deterministic systems are considered equivalent if they generate the same language. Most results are stated in terms of languages. Automata are only used to show how results can be computed. This has the advantage that the results are not dependent on the automaton representation but also hold for other representations such as Petri nets and process algebras. In this thesis we want to extend the results to nondeterministic discrete event systems. It will be shown that the language alone is not sufficient to describe the behavior of a nondeterministic discrete event system.

Example 2.4 Consider a vending machine that hands out a cookie or a chocolate bar in exchange for a coin. In Figure 2.2 the representations of three vending machines are given by finite state automata. All three machines can generate the same language but will behave differently. Therefore, it is not sufficient to describe their behavior by the language that they can generate. How the machines behave is best illustrated by letting a user operate the machines.

After a client inserts a coin, the first machine will always hand out what the user requests. It will never refuse to give a cookie nor a chocolate bar. If a customer insists on having a chocolate bar from the second machine and this machine is in the state in which it can only hand out a cookie, then no event can be executed and the system is said to be in *deadlock*. The machine will however never refuse to hand out a cookie. The third machine can sometimes refuse to give a cookie and sometimes refuse to give a chocolate bar. But it cannot refuse both at the same time. If a user requests either of the sweets, no matter if it is a cookie or chocolate bar, then the machine cannot refuse and it must hand out one of them.

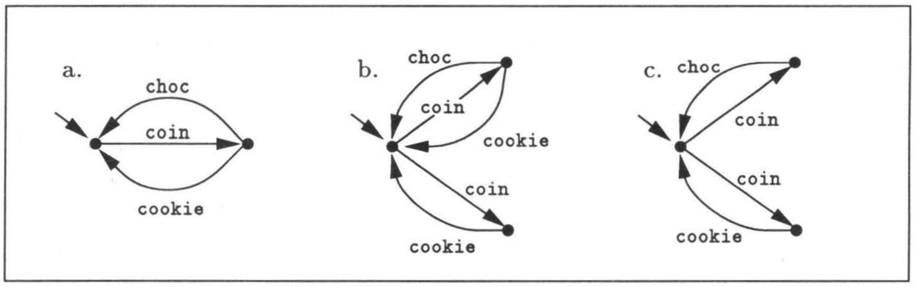


Figure 2.2: Models of a vending machine.

To describe the behavior of the machines it is necessary to not only describe the events that can be executed, i.e. the language, but also the event sets that can be refused. This is the basis of failure semantics [23]. A machine can refuse event set $R \subseteq \Sigma$ after string s, if it can reach a state by executing string s, and it cannot execute any event of event set R in this state. The event sets that can be refused are called *refusals*. A set of refusals is called a *refusal set*. Let $\text{ref}(X, s)$ denote the refusal set of automaton X after it has executed trace s. Then

$$R \in \text{ref}(X,s) \quad \Longleftrightarrow \quad \exists q \in \delta(X,s) \text{ s.t. } \forall \sigma \in R \;\; \delta(X,q,\sigma) = \emptyset. \tag{2.1}$$

For instance, the refusal set of the third machine after a coin is inserted is the following.

$$\{\emptyset, \{\texttt{coin}\}, \{\texttt{cookie}\}, \{\texttt{choc}\}, \{\texttt{coin}, \texttt{cookie}\}, \{\texttt{coin}, \texttt{choc}\}\}.$$

As explained in Example 2.4 the machine cannot refuse both the cookie and the chocolate bar, so the event set $\{\texttt{cookie}, \texttt{choc}\}$ is not an element of the refusal set.

In the same way as languages are used to model the behavior of deterministic systems, will the combination of languages and refusal sets be used to model the behavior of nondeterministic systems. Automata will only be used as representation of systems and to perform computations.

Definition 2.5 A *process* is a triple $A = \big(\Sigma(A), \mathrm{L}(A), \mathrm{ref}(A)\big)$, where

$\Sigma(A) \subseteq \Sigma$ is the set of event labels,
$\mathrm{L}(A) \subseteq \Sigma(A)^*$ is the language generated by A,
for $s \in \mathrm{L}(A)$, $\mathrm{ref}(A,s) \subseteq 2^{\Sigma(A)}$ is the refusal set after s,

and which satisfies the following five conditions:

i) $\varepsilon \in \mathrm{L}(A)$,
ii) $\mathrm{L}(A) = \overline{\mathrm{L}(A)}$,
iii) $s \in \mathrm{L}(A) \Rightarrow \emptyset \in \mathrm{ref}(A,s)$,
iv) $s \in \mathrm{L}(A) \wedge R \in \mathrm{ref}(A,s) \wedge R' \subseteq R \Rightarrow R' \in \mathrm{ref}(A,s)$,
v) $s \in \mathrm{L}(A) \wedge R \in \mathrm{ref}(A,s) \Rightarrow R \cup \rho(\mathrm{L}(A),s) \in \mathrm{ref}(A,s)$.

These conditions state respectively that the language has to be nonempty and prefix closed, the refusal sets have to be nonempty and closed under the operation of taking the subset, and events that cannot be refused must be in the language [23].

For $s \notin \mathrm{L}(A)$ the refusal set $\mathrm{ref}(A,s)$ is defined to be $2^{\Sigma(A)}$. Let $\Pi(\Sigma)$ be the set of all processes A with $\Sigma(A) = \Sigma$.

It can be shown that there exists an automaton which represents a given $(\Sigma, \mathrm{L}, \mathrm{ref})$-triple if and only if the triple satisfies the conditions *i*–*v*.

Sometimes we will refer to the $(\Sigma, \mathrm{L}, \mathrm{ref})$-triple with empty language (and $\mathrm{ref}(.,s) = 2^{\Sigma}$ for all $s \in \Sigma^*$) as the *empty process*, although it is officially not a process, because it violates condition *i*.

The ref-function associates to each string a set of subsets of Σ. If a subset R is an element of $\mathrm{ref}(A,s)$ then the process has the possibility after trace s to block all events in R. That is, if a user offers (via the synchronous composition defined below) to the system a set of events, which is in the refusal set, then the system has the possibility to block all these events. No event can be executed. This is called a deadlock

Definition 2.6 System A can *deadlock* after trace $s \in \mathrm{L}(A)$ if $\Sigma(A) \in \mathrm{ref}(A,s)$. System A is *deadlock-free* after s if $\Sigma(A) \notin \mathrm{ref}(A,s)$.

It will be assumed that if A is deadlock-free after trace s then eventually it will execute an event from $\lambda(\mathrm{L}(A),s)$. So a system will continue unless it deadlocks. Note however, that if a process *can* deadlock after a trace then this does not mean that it actually *will* deadlock. If a process can deadlock after trace s, then, according to 2.1, it can reach a state q_1 in which it cannot execute any further event. But it could be, because of nondeterminism, that it can also reach another state, say q_2, in which it *can* execute an event.

Deterministic Processes

In Section 2.1 automaton X is called deterministic if from each observation $s \in \mathrm{L}(X)$ it is uniquely determined in which state system X is. So it is also uniquely determined which events can be executed, and which events can be refused after s. A process will be called deterministic if any event that can be executed after a trace cannot be refused after the same trace.

Definition 2.7 Process A is called *deterministic* if for all $s \in \mathrm{L}(A)$,

$$R \in \mathrm{ref}(A, s) \iff R \subseteq \rho(\mathrm{L}(A), s).$$

The class of deterministic processes does not correspond exactly with the class of deterministic automata. Some nondeterministic automata have a deterministic process representation. Consider for instance the automaton shown in Figure 2.3. Although the automaton is nondeterministic by the definition given in Section 2.1, its behavior is clearly deterministic. After any observation it is clear which events can be executed and which will be refused.

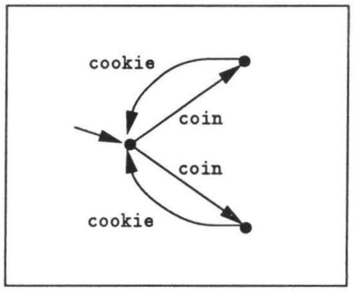


Figure 2.3: A nondeterministic vending machine?

In the sequel we will call a discrete event system deterministic if its process representation is deterministic.

It was shown in Section 2.1 that a nonempty prefix closed language $K \subseteq \Sigma^*$ defines a deterministic automaton, which is called the canonical automaton representation of language K. The following construction gives the process representation of this deterministic system.

$$\mathrm{Det}(K) = (\Sigma,\ K,\ \mathrm{ref}(K)),$$

where

$$\mathrm{ref}(K, s) \quad = \quad 2^{\rho(K,s)}, \quad \text{for all } s \in K.$$

Proposition 2.8 *Let $K \subseteq \Sigma^*$ be a nonempty prefix closed language. Then* $\mathrm{Det}(K) \in \Pi(\Sigma)$.

Proof. We have to prove that Det(K) satisfies points *i* – *v* of Definition 2.5. As $\mathrm{L}(\mathrm{Det}(K)) = K$ and K is nonempty and prefix closed, it automatically follows that Det(K) satisfies points *i* and *ii*. Point *iii* is satisfied because $\emptyset \subseteq \rho(K, s)$ for all $s \in K$. Points *iv* and *v* follow directly from the construction of $\mathrm{ref}(\mathrm{Det}(K), s)$. □

Operations on Processes

Control will be enforced by synchronization on common events. The controlled system (i.e. the synchronous composition of the plant and the supervisor) can only execute those events that both the supervisor and the plant can execute. Below the synchronous composition of two processes with equal alphabets is defined. The synchronous composition of two processes with different alphabets is defined in Section 5.1.

Definition 2.9 Let A and B be two processes with the same alphabet. The *synchronous composition* of processes A and B, denoted $A||B$, is defined by the following relations.

$$\begin{aligned}
\Sigma(A||B) &= \Sigma(A) = \Sigma(B),\\
\mathrm{L}(A||B) &= \mathrm{L}(A) \cap \mathrm{L}(B),\\
\mathrm{ref}(A||B, s) &= \{R_{\mathrm{a}} \cup R_{\mathrm{b}} \subseteq \Sigma(A||B) : R_{\mathrm{a}} \in \mathrm{ref}(A, s),\ R_{\mathrm{b}} \in \mathrm{ref}(B, s)\}.
\end{aligned}$$

It is not difficult to show that for processes A, $B \in \Pi(\Sigma)$ the synchronous product $A||B$ is also a process. That is, $\Pi(\Sigma)$ is closed under synchronous composition.

The nondeterministic choice between processes A and B is the process $A \sqcup B$, that behaves either as process A or as process B. The selection between them is made internally inside the process $A \sqcup B$. The environment of process $A \sqcup B$ can neither observe nor influence this selection.

Definition 2.10 Let A and B be two processes with the same alphabet. The *nondeterministic choice* between A and B, denoted by $A \sqcup B$, is defined by the following equations.

$$\begin{aligned}
\Sigma(A \sqcup B) &= \Sigma(A) = \Sigma(B),\\
\mathrm{L}(A \sqcup B) &= \mathrm{L}(A) \cup \mathrm{L}(B),\\
\mathrm{ref}(A \sqcup B, s) &= \begin{cases} \mathrm{ref}(A, s), & \text{if } s \in \mathrm{L}(A) - \mathrm{L}(B),\\ \mathrm{ref}(B, s), & \text{if } s \in \mathrm{L}(B) - \mathrm{L}(A),\\ \mathrm{ref}(A, s) \cup \mathrm{ref}(B, s), & \text{if } s \in \mathrm{L}(A) \cap \mathrm{L}(B). \end{cases}
\end{aligned}$$

If A and B are processes then so is $A \sqcup B$. Let $\mathcal{A}$ be a possibly infinite set of processes, with all elements having the same alphabet. Let $\Sigma(\mathcal{A})$ be this alphabet. Then $\bigsqcup \mathcal{A}$ is the nondeterministic choice of all processes in A.

$$
\begin{aligned}
\Sigma(\bigsqcup\mathcal{A}) &= \Sigma(\mathcal{A}),\\
\mathrm{L}(\bigsqcup\mathcal{A}) &= \bigcup\{\mathrm{L}(A) : A \in \mathcal{A}\},\\
\mathrm{ref}(\bigsqcup\mathcal{A}, s) &= \bigcup\{\mathrm{ref}(A, s) : A \in \mathcal{A} \text{ s.t. } s \in \mathrm{L}(A)\}, \qquad \forall s \in \mathrm{L}(\bigsqcup\mathcal{A}).
\end{aligned}
$$

If all elements of $\mathcal{A}$ are processes then so is $\bigsqcup\mathcal{A}$.

A process controlling process $A \sqcup B$ must be able to control both processes without knowing which process is selected.

2.3. Behavior State Representation

For processes there is no standard, well established, automaton representation such as the canonical automaton representation used for languages. In this section we will define a representation that will be used in the rest of the thesis as automaton representation of processes. It will be used to describe processes and to perform computations on processes. The representation is based on an equivalence relation similar to the Nerode equivalence relation used for languages [26].

Let A/s be the process that behaves as process A after it has executed trace $s \in \Sigma(A)$.

$$A/s \;=\; (\Sigma(A),\ \mathrm{L}(A)/s,\ \mathrm{ref}(A/s)),$$

where

$$\mathrm{ref}(A/s, v) \;=\; \mathrm{ref}(A, sv) \qquad \text{for all } v \in \Sigma(A)^*.$$

Note that if $s \notin \mathrm{L}(A)$ then A/s is the empty process.

In the same way as is done with Nerode equivalence for languages, consider traces s and s' equivalent if $A/s = A/s'$.

$$s \equiv_A s' \quad\Longleftrightarrow\quad A/s = A/s'.$$

Let $[s]_A$ denote the equivalence set induced by this equivalence relation containing trace s.

$$[s]_A \;=\; \{s' \in \Sigma^* : A/s = A/s'\}.$$

One can regard $[s]_A$ as the state reached after trace s. To differentiate this notion of state from the states used in regular nondeterministic automata, we will call $[s]_A$ the *behavior state* reached after trace s.

Note that $[s]_A$ is also defined for traces $s \notin \mathrm{L}(A)$. It can be shown that if traces s and s' are not in the language of A, then $[s]_A = [s']_A$.

Lemma 2.11 *Let $s, s' \in \Sigma(A)^*$, $s, s' \notin \mathrm{L}(A)$. Then $[s]_A = [s']_A$.*

Proof. Let $s \in \Sigma(A)^* - \mathrm{L}(A)$. Then $\mathrm{L}(A/s) = \{v \in \Sigma(A)^* : sv \in \mathrm{L}(A)\} = \emptyset$, and, according to Definition 2.5, $\mathrm{ref}(A/s, v) = 2^{\Sigma(A)}$ for all $v \in \Sigma(A)^*$. So, for all $s' \in \Sigma(A)^* - \mathrm{L}(A)$, $\mathrm{L}(A/s) = \emptyset = \mathrm{L}(A/s')$, and for all $v \in \Sigma(A)^*$, $\mathrm{ref}(A/s, v) = 2^{\Sigma(A)} = \mathrm{ref}(A/s', v)$. Hence $[s]_A = [s']_A$. □

Let the *dump state* be the equivalence set containing all traces that are not in the language of A.

Lemma 2.12 *Let $s, s' \in \Sigma(A)^*$ such that $[s]_A = [s']_A$. Then*

$$s \in \mathrm{L}(A) \iff s' \in \mathrm{L}(A)$$

Proof. If $s \in \mathrm{L}(A)$ then $\varepsilon \in \mathrm{L}(A/s)$. So, by $[s]_A = [s']_A$, $\varepsilon \in \mathrm{L}(A/s')$. This implies that $s' \in \mathrm{L}(A)$. The converse holds by symmetry. □

Definition 2.13 The *behavior state representation* of process A is a basic deterministic automaton extended with an extra element. This extra element is denoted $\mathrm{ref}(A)$. It maps behavior states to the corresponding refusal sets. The behavior state representation of process A is defined by

$$\big(\Sigma(A),\ Q(A),\ \delta(A),\ Q_0(A),\ \mathrm{ref}(A)\big),$$

where

$$\begin{aligned} Q(A) &= \{[s]_A : s \in \mathrm{L}(A)\}, \\ Q_0(A) &= [\varepsilon]_A, \end{aligned}$$

and for all $[s]_A \in Q(A)$

$$\begin{aligned} \delta(A, [s]_A, \sigma) &= \begin{cases} [s\sigma]_A, & \text{if } \sigma \in \lambda(\mathrm{L}(A), s), \\ \emptyset & \text{otherwise,} \end{cases} \\ \mathrm{ref}(A, [s]_A) &= \mathrm{ref}(A, s). \end{aligned}$$

Note that $Q(A)$ does not contain the dump state. Occasionally, also the behavior state space including the dump state will be used. Define

$$Q^+(A) \quad = \quad \{[s]_A : s \in \Sigma^*\}.$$

Although process A may be nondeterministic, the behavior state representation is always a deterministic automaton. It can be seen as if the nondeterministic properties of process A are encoded inside the refusal sets of the behavior states instead of modeled by the transition function.

In Figure 2.4 the behavior state representation of the process shown in Figure 2.2.c is given. For compactness reasons, only the maximal refusals, i.e. the refusals not strictly contained in another refusal, are shown. As refusal sets are closed under the operation of taking subsets, the whole refusal set can

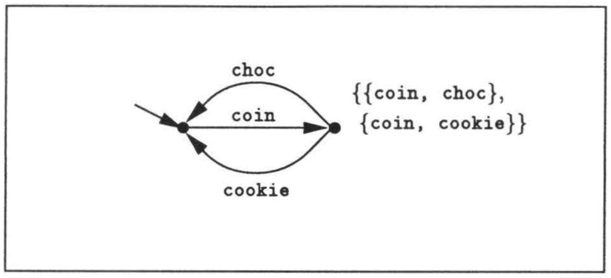


Figure 2.4: Behavior state representation of a vending machine.

be derived from the maximal refusals. Also the refusal sets of states $[s]_A$ with $\mathrm{ref}(A, [s]_A) = 2^{\rho(\mathrm{L}(A),s)}$ are not shown. These refusals can be derived from the outgoing arrows of the state.

The refusal set of state $[\varepsilon]_A$ is

$$\mathrm{ref}(A, [\varepsilon]_A) \;=\; 2^{\rho(\mathrm{L}(A),\varepsilon)} \;=\; \{\emptyset,\, \{\mathtt{choc}\},\, \{\mathtt{cookie}\},\, \{\mathtt{choc}, \mathtt{cookie}\}\}.$$

The refusal set of state $[\mathtt{coin}]_A$ is the set of all subsets of $\{\mathtt{coin}, \mathtt{cookie}\}$ and $\{\mathtt{coin}, \mathtt{choc}\}$. It is shown in Section 2.2.

Converting a nondeterministic finite state machine to a behavior state representation is basically the same as converting a nondeterministic state machine to a deterministic version. This conversion has a known complexity that is worst case exponential in the size of the state space of the original state machine. But in practice systems have sufficient structure, such that this conversion may not be a problem.

2.4. Specification, Implementation, and Control

In general a design problem can be defined as: given a specification, find an implementation that satisfies the specification. A design problem can be considered a supervisory control problem if the implementation consists of an already existing uncontrolled process G and a still to be designed supervisor process S. In this section we will define the control problem of finding a supervisor S such that $G||S$ can replace a given specification process E. The following example will illustrate in what sense an implementation must be able to replace a specification.

Example 2.14 A system usually does not work on its own. It is embedded in a larger system. For instance a hard-disk unit is used inside a computer system. The computer is usually designed in a different place than the hard-disk unit. During the design phase a standard is negotiated between the computer manufacturer and the disk manufacturer. This standard is the specification of the hard-disk. After this standard is established the computer designer models a computer system in which it expects a hard-disk unit that behaves according to this specification. It is the hard-disk developers task to build a hard-disk unit that satisfies this specification. Without him knowing how the computer system will look, he has to design a unit that works together with this system.

Consider the following implementation relation [15, 23] .

Definition 2.15 Let $A, B \in \Pi(\Sigma)$. *A reduces B*, denoted by $A \precsim B$, if

$$\begin{array}{lll} i) & \mathrm{L}(A) \subseteq \mathrm{L}(B), & \text{and} \\ ii) & \mathrm{ref}(A,s) \subseteq \mathrm{ref}(B,s) & \text{for all } s \in \mathrm{L}(A). \end{array}$$

Here, point *i* states that system A may only do what system B allows, and point *ii* states that A may only refuse what B can also refuse. We will say that process $G||S$ implements specification E if $G||S \precsim E$.

Note that $A = B$ if and only if $A \precsim B$ and $B \precsim A$. This property will be used in many proofs. The next result is well known from computer science [10]. It states that the reduction relation forms a congruence with the synchronous composition.

Proposition 2.16 *Let $A_1, A_2, B_1, B_2 \in \Pi(\Sigma)$ such that $A_1 \precsim A_2$ and $B_1 \precsim B_2$. Then $A_1||B_1 \precsim A_2||B_2$.*

In Example 2.14 the implementation of the hard-disk has to be such that it can replace its specification in any computer system. This is guaranteed by the reduction relation. Let $G||S$ stand for the implementation of the hard-disk, E for the specification, and C for the rest of the computer system. Then the following implication, which is a direct consequence of Proposition 2.16, states that $G||S$ can replace E in any computer system.

$$G||S \precsim E \quad \Rightarrow \quad \forall C,\ (G||S)||C \precsim E||C. \tag{2.2}$$

This implication shows that the reduction relation is strong enough to use it as an implementation relation. The following result shows that it also forms a necessary condition to guarantee deadlock-free behavior. This result forms the main motivation for the use of failure semantics and the reduction relation.

Theorem 2.17 *Let $A, E \in \Pi(\Sigma)$.*

$$\begin{array}{cc} & A \precsim E \\ & \Longleftrightarrow \\ \forall C \in \Pi(\Sigma) & \mathrm{L}(A||C) \subseteq \mathrm{L}(E||C), \ \textit{and} \\ & E||C \textit{ deadlock-free} \Rightarrow A||C \textit{ deadlock-free.} \end{array}$$

Proof. The $\Rightarrow$-part follows from Proposition 2.16. For the proof of the $\Leftarrow$-part assume that A does not reduce E. Then either $\mathrm{L}(A) \not\subseteq \mathrm{L}(E)$ or there exists an $s \in \mathrm{L}(A)$ such that $\mathrm{ref}(A,s) \not\subseteq \mathrm{ref}(E,s)$. Assume there exists an $s \in \mathrm{L}(A)$ such that $s \notin \mathrm{L}(E)$. Let C be a process such that $s \in \mathrm{L}(C)$. Then $s \in \mathrm{L}(A||C)$ but $s \notin \mathrm{L}(E||C)$, so $\mathrm{L}(A||C) \not\subseteq \mathrm{L}(E||C)$. For the other alternative let $s \in \mathrm{L}(A)$ such that there exists an $R \in \mathrm{ref}(A,s)$ and $R \notin \mathrm{ref}(E,s)$. Let C be a process such that $\mathrm{ref}(C,s) = 2^{\Sigma - R}$. Then $\Sigma = R \cup (\Sigma - R) \in \mathrm{ref}(A||C,s)$, but $\Sigma \notin \mathrm{ref}(E||C,s)$. So $E||C$ is deadlock-free, but $A||C$ is not. □

The basic supervisory control problem can be formulated as follows. Given an uncontrolled system G and a specification E, find a supervisor S such that $G||S \precsim E$.

In some applications the supervisor does not have the ability to block all events. For instance if an alarm event is executed when some water level exceeds a threshold, then this event can be observed by the supervisor but it cannot be blocked. If this event has to be prevented from occurring then somewhere else in the system some other events have to be blocked (for instance the event corresponding with the closing of a waste gate) such that the alarm event cannot be executed.

Usually the presence of uncontrollable events is modeled by splitting up the event set Σ into controllable and uncontrollable events, Σ_{c} and Σ_{uc} respectively. A supervisor is called complete if it does not block any uncontrollable events.

Definition 2.18 Supervisor S is *complete* (w.r.t process G) if

$$\forall s \in \mathrm{L}(G||S),\ \forall R_s \in \mathrm{ref}(S,s),\ R_s \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s).$$

Definition 2.19 Let the uncontrolled system $G \in \Pi(\Sigma)$ and a specification $E \in \Pi(\Sigma)$ be given. The *basic supervisory control problem* is to find a complete supervisor $S \in \Pi(\Sigma)$, such that $G||S \precsim E$.

2.5. Comparison of Frameworks

In this section we will compare the approach based on failure semantics and the reduction relation with other approaches. The comparison is not intended to be complete. It just illustrates some differences between the approaches.

The original framework introduced by Ramadge and Wonham [52] was intended to handle only deterministic systems. The framework presented in this thesis is also capable of handling nondeterministic systems. But even if we restrict our attention to deterministic systems there are some important differences.

It can be shown in the framework presented by Ramadge and Wonham that the corresponding implication of (2.2) is not satisfied. In that framework a discrete event system, A, is modeled by the triple $(\Sigma(A), \mathrm{L}(A), \mathrm{L}_{\mathrm{m}}(A))$, where

$\mathrm{L}(A) \subseteq \Sigma(A)^*$ is the prefix closed language that A can generate and $\mathrm{L_m}(A) \subseteq \mathrm{L}(A)$ is the language that A accepts or marks.

Definition 2.20 Let A and B be discrete event systems. $A \sqsubseteq_{\mathrm{m}} B$ if

i) $\mathrm{L}(A) \subseteq \mathrm{L}(B)$,
ii) $\mathrm{L_m}(A) \subseteq \mathrm{L_m}(B)$,
iii) $\mathrm{L}(A) = \overline{\mathrm{L_m}(A)}$.

A system is called *M-nonblocking* if it satisfies point *iii*.

System $G||S$ is considered an implementation of E in the Ramadge Wonham framework if $G||S \sqsubseteq_{\mathrm{m}} E$. Note that if $\mathrm{L}(E) = \overline{\mathrm{L_m}(E)}$ then point *ii* and *iii* together imply point *i*. Usually the specification is not given as a process but as a language $K \subseteq \mathrm{L_m}(G)$. In this case the specification process E can be defined by $\mathrm{L}(E) = \overline{K}$ and $\mathrm{L_m}(E) = K$. Sometimes a non-marking supervisor is required, that is $\mathrm{L_m}(S) = \mathrm{L}(S)$. In this case it is usually assumed that $\mathrm{L_m}(E) = \mathrm{L}(E) \cap \mathrm{L_m}(G)$. These differences are not important for the following discussion, which mainly concerns point *iii*.

We want that an implementation can replace the specification in any environment. This is however not guaranteed by the $\sqsubseteq_{\mathrm{m}}$ relation. The next example illustrates that in general it cannot be guaranteed that the implementation is M-nonblocking in any environment, i.e.

$$G||S \sqsubseteq_{\mathrm{m}} E \wedge E||C \text{ is M-nonblocking } \not\Rightarrow (G||S)||C \text{ is M-nonblocking.}$$

Example 2.21 Let E be the specification with $\mathrm{L_m}(E) = \mathsf{a}(\mathsf{b}+\mathsf{c})$ and $\mathrm{L}(E) = \overline{\mathsf{a}(\mathsf{b}+\mathsf{c})}$. Let $A = G||S$ be the implementation with $\mathrm{L_m}(A) = \mathsf{ab}$ and $\mathrm{L}(A) = \overline{\mathsf{ab}}$. And let C represent the rest of the computer system with $\mathrm{L_m}(C) = \mathsf{ac}$ and $\mathrm{L}(C) = \overline{\mathsf{ac}}$. Observe that $A \sqsubseteq_{\mathrm{m}} E$, but $\mathrm{L}(A||C) = \mathsf{a}$ and $\mathrm{L_m}(A||C) = \varepsilon$, thus $\mathrm{L}(A||C) \neq \overline{\mathrm{L_m}(A||C)}$. So $A||C \not\sqsubseteq_{\mathrm{m}} E||C$.

It can be derived from results obtained by Wonham and Ramadge [63] on modular control that $(G||S)||C$ is only M-nonblocking if $\mathrm{L_m}(G||S)$ and $\mathrm{L_m}(C)$ are *non-conflicting*. That is, processes A and B are non-conflicting if common prefixes in both processes can be extended to a common marked trace.

$$\overline{\mathrm{L_m}(A) \cap \mathrm{L_m}(B)} = \overline{\mathrm{L_m}(A)} \cap \overline{\mathrm{L_m}(B)}.$$

This constraint also limits the use of modular control in the Ramadge Wonham framework. If a specification E can be decomposed as $E_1||E_2 = E$, then it has computational advantages to first synthesize both S_1 and S_2 such that $G||S_1$ implements E_1 and $G||S_2$ implements E_2. In the framework based on failure semantics it can be deduced from Proposition 2.16 and the fact that $G \precsim G||G$, that

$$G||S_1 \precsim E_1 \ \wedge \ G||S_2 \precsim E_2 \ \Rightarrow \ G||S_1||S_2 \precsim E_1||E_2.$$

In the Ramadge Wonham framework however it is necessary that $\mathrm{L_m}(G||S_1)$ and $\mathrm{L_m}(G||S_2)$ are non-conflicting in order to guarantee that $G||S_1||S_2$ is M-nonblocking. This constraint is often not easy to satisfy.

The discussion above considers how well the M-nonblocking property and the deadlock freeness property behave within their own framework. It does not compare the properties directly with each other. The M-nonblocking property states that a process is always able to complete a task, whereas the deadlock freeness property states that a process is always able to continue. Note that it cannot be specified by a marked language that the implementation should be deadlock free. Even if there are transitions leading out of each marked state in the specification, then still an implementation which deadlocks in a marked state satisfies the specification according to Definition 2.20. So marking cannot be used to guarantee deadlock free behavior. It depends on the particular application which approach is more suited.

The marking condition on states can be replaced by an event that indicates the completion of a task. This issue is treated in Section 4.6. With this approach a process can be considered nonblocking if it cannot refuse such a task completion event. The nonblocking property can then be adequately handled within the framework based on failure semantics.

Within the computer science area synthesis is investigated based on infinite trace theory [18, 38, 48]. Also within the control theory area this approach has been followed [58]. Infinite trace automata have an acceptance condition, which is similar to the marking condition for finite trace automata. Because of this acceptance condition the corresponding implication of (2.2) will not be satisfied within this framework. Also, there will be extra constraints necessary for modular control synthesis.

Nondeterministic Specifications

It is logical, if one considers that the implementation should be able to replace the specification, that the specification is given as a process. In the rest of this section it will be shown that this specification method has more expressive power than a specification given as a language or as a range of languages.

Initially in discrete event control the problem was posed to find a supervisor S such that

$$\mathrm{L}(G||S) \quad = \quad \mathrm{L}(E).$$

Conditions were found under which such a solution exists. But this control problem formulation is rather rigid. It does not allow for any flexibility. Therefore the specification was considered to denote the set of all legal traces and the following more flexible control problem was posed. Find a supervisor S such that

$$\mathrm{L}(G||S) \quad \subseteq \quad \mathrm{L}(E).$$

The task was to find the largest solution of this control problem. However, this control problem formulation allowed to many implementations. For instance the language $\{\varepsilon\}$ also satisfies the inclusion relation. Two solutions were presented to restrict the class of legal implementation. The one is marking, which we discussed above and will not be considered here. The other is the minimal allowable language, usually denoted $\mathrm{L}(A)$. The control problem with the minimal allowable language is to find a supervisor S such that

$$\mathrm{L}(A) \quad \subseteq \quad \mathrm{L}(G||S) \quad \subseteq \quad \mathrm{L}(E)$$

The minimal allowable language is also used to synthesize a solution if the supremal solution cannot be synthesized. However, it is not always possible to define an adequate minimal allowable language.

Example 2.22 Consider a car. If one wants to describe how a car should be operated, one should include cars with a manually operated gearbox, and cars with an automatic transmission. As the intersection of the language needed to describe the automatic transmission, and the language needed to describe the manual gearbox is empty, the minimal allowable language will be empty as well. But cars do need a mechanism to drive their wheels. Thus, a minimal allowable language cannot describe all minimal allowable behavior. If one wants to specify a minimal allowable language, then one already has to make the choice whether the car will have a manual gearbox or an automatic transmission. So, one already has to decide on some design issues in the minimal allowable language. In a control context this would mean that the minimal allowable language already specifies some parts of the solution to the control problem.

Using a nondeterministic specification, both options can be included by allowing the nondeterministic choice between the automatic transmission and the manual gearbox. The specification process can be seen as a representation of the set of all legal implementations.

Chapter 3
Controller Synthesis

In this chapter we will present a method to derive solutions for the basic supervisory control problem. The solution scheme will also be used in the following chapters. As with supervisory control based on languages, the synthesis method relies on a underlying complete lattice structure. See [7] for an introduction to lattice theory.

In the following section a supervisor will be defined, called the supremal supervisor, which is the nondeterministic choice between all solutions to the basic supervisory control problem. This supremal supervisor can be used in subsequent steps to derive solutions that have to satisfy other constraints as well. In Section 3.3 the deterministic least restrictive solution will be derived in this way.

The concept of a supremal supervisor which describes the set of all solutions to a supervisory control problem is very useful. It divides difficult control problems into smaller steps. In the first step the supremal supervisor is generated which describes all complete supervisors S such that $G||S \underset{\sim}{\sqsubset} E$. In the following steps the supremal supervisor will serve as a specification. It will be shown that a supervisor which implements the supremal supervisor is automatically complete and reduces E in combination with G. In the following steps completeness, system G, and system E do not have to be considered. The supremal supervisor contains all information necessary to find a solution. This simplifies the subsequent steps of the control problem.

3.1. Supremal Supervisor

In [10, Theorem 1] it is proven that $\Pi(\Sigma)$ is a complete upper semi-lattice with the reduction relation as partial ordering and the nondeterministic choice as join operator.[1] Completeness of $\Pi(\Sigma)$ implies that the supremal element (= the least upper bound) of any nonempty subset of processes is a process. Completeness does not imply that the supremal is an element of the subset. It only implies that the supremal exists and is a process, i.e. that it satisfies the properties of Definition 2.5. The supremal of a nonempty set of processes is equal to the nondeterministic choice of all elements in this subset.

Definition 3.1 Let $\mathcal{C}(G,E)$ be the set of all supervisors that solve the basic supervisory control problem.

$$\mathcal{C}(G,E) \quad = \quad \{S \in \Pi(\Sigma) : G||S \sqsubseteq_{\sim} E \text{ and } S \text{ is complete}\}.$$

If $\mathcal{C}(G,E)$ is nonempty then the *supremal supervisor*, denoted $S^{\uparrow}$, is the supremal of $\mathcal{C}(G,E)$ with respect to the reduction ordering relation.

$$\begin{aligned} S^{\uparrow} &= \sup_{\underset{\sim}{\sqsubset}} \mathcal{C}(G,E) \\ &= \bigsqcup \{S \in \Pi(\Sigma) : G||S \underset{\sim}{\sqsubset} E \text{ and } S \text{ is complete}\}. \end{aligned}$$

If $\mathcal{C}(G,E)$ is empty then let $S^{\uparrow}$ be the empty process.

Theorem 3.2 *Let $G, E, S \in \Pi(\Sigma)$ and let $S^{\uparrow}$ be as defined above.*

$$S \underset{\sim}{\sqsubset} S^{\uparrow} \quad \Longleftrightarrow \quad G||S \underset{\sim}{\sqsubset} E \text{ and } S \text{ is complete.}$$

Proof. The $\Leftarrow$-part of the theorem follows from the supremality of $S^{\uparrow}$.

($\Rightarrow$-part.) If $\mathcal{C}(G,E)$ is empty then $S^{\uparrow}$ is empty. No process reduces $S^{\uparrow}$, so the condition in the theorem is satisfied. For the rest of the proof let $\mathcal{C}(G,E)$ be nonempty. Let $S \in \Pi(\Sigma)$, $S \underset{\sim}{\sqsubset} S^{\uparrow}$. We have to prove that $G||S \underset{\sim}{\sqsubset} E$ and S is complete. From the definition of nondeterministic choice it can be derived that

$$\mathrm{L}(S^{\uparrow}) \quad = \quad \bigcup\{\mathrm{L}(S') : S' \in \mathcal{C}(G,E)\},$$

and that for all $s \in \mathrm{L}(S^{\uparrow})$

$$\mathrm{ref}(S^{\uparrow}, s) \quad = \quad \bigcup\{\mathrm{ref}(S', s) : S' \in \mathcal{C}(G,E) \text{ s.t. } s \in \mathrm{L}(S')\}.$$

First it will be proven that $\mathrm{L}(G||S) \subseteq \mathrm{L}(E)$.

[1]Note that the ordering used by Hoare et al. is equal to the reduction ordering, except that processes are ordered in the opposite direction.

$$\begin{array}{lll}
s \in \mathrm{L}(G||S) & \Rightarrow & s \in \mathrm{L}(G) \wedge s \in \mathrm{L}(S) \\
& \Rightarrow & s \in \mathrm{L}(G) \wedge s \in \mathrm{L}(S^{\uparrow}) \\
& \Rightarrow & s \in \mathrm{L}(G) \wedge s \in \bigcup\{\mathrm{L}(S') : S' \in \mathcal{C}(G,E)\} \\
& \Rightarrow & s \in \mathrm{L}(G) \wedge \exists S' \in \mathcal{C}(G,E) \text{ s.t. } s \in \mathrm{L}(S') \\
& \Rightarrow & \exists S' \in \mathcal{C}(G,E) \text{ s.t. } s \in \mathrm{L}(G||S') \\
& \Rightarrow & \hfill [\mathrm{L}(G||S') \subseteq \mathrm{L}(E)] \\
& & s \in \mathrm{L}(E).
\end{array}$$

So $\mathrm{L}(G||S) \subseteq \mathrm{L}(E)$. It can be derived from Definition 2.9 that for all $s \in \mathrm{L}(G||S)$

$$\begin{array}{c}
\mathrm{ref}(G||S, s) \subseteq \mathrm{ref}(E, s) \\
\Longleftrightarrow \\
\forall R_s \in \mathrm{ref}(S,s),\ \forall R_g \in \mathrm{ref}(G,s)\ R_s \cup R_g \in \mathrm{ref}(E,s)
\end{array}$$

Then

$$\begin{array}{lll}
R_s \in \mathrm{ref}(S,s) & \Rightarrow & R_s \in \mathrm{ref}(S^{\uparrow}, s) \\
& \Rightarrow & R_s \in \bigcup\{\mathrm{ref}(S',s) : S' \in \mathcal{C}(G,E) \text{ s.t. } s \in \mathrm{L}(S')\} \\
& \Rightarrow & \exists S' \in \mathcal{C}(G,E) \text{ s.t. } s \in \mathrm{L}(S') \text{ and } R_s \in \mathrm{ref}(S',s) \\
& \Rightarrow & \hfill [\mathrm{ref}(G||S',s) \subseteq \mathrm{ref}(E,s)] \\
& & \forall R_g \in \mathrm{ref}(G,s)\ R_s \cup R_g \in \mathrm{ref}(E,s).
\end{array}$$

So for all $s \in \mathrm{L}(G||S)$, $\mathrm{ref}(G||S,s) \subseteq \mathrm{ref}(E,s)$. Hence $G||S \sqsubseteq E$. As S' is complete it follows that $R_s \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s)$, so S is also complete. It can be concluded that S solves the basic supervisory control problem. □

As $S^{\uparrow} \sqsubseteq S^{\uparrow}$, it follows that $G||S^{\uparrow} \sqsubseteq E$ and $S^{\uparrow}$ is complete. So if $\mathcal{C}(G,E)$ is nonempty then $S^{\uparrow} \in \mathcal{C}(G,E)$.

Note that the corresponding relation of Theorem 3.2 in a language semantics setting does not hold. Let

$$S_{\mathrm{L}}^{\uparrow} \;=\; \sup_{\subseteq}\{S : \mathrm{L}(G||S) \subseteq \mathrm{L}(E) \text{ and } S \text{ complete}\}.$$

Then

$$\mathrm{L}(S) \subseteq \mathrm{L}(S_{\mathrm{L}}^{\uparrow}) \;\not\Rightarrow\; S \text{ complete}.$$

That is, it is not guaranteed by the inclusion relation that S does not block any uncontrollable events. In failure semantics this constraint is incorporated in the refusal sets of $S^{\uparrow}$.

3.2. Construction of the Supremal Supervisor

Let system H be given as a $(\Sigma, \mathrm{L}, \mathrm{ref})$ triple, but not necessarily a process. Define $\Pi(H)$ as the set of processes that reduce H. Let $\Pi^{\uparrow}(H)$ be the supremum of this set. If $\Pi(H)$ is empty then $\Pi^{\uparrow}(H)$ is empty.

$$\begin{aligned} \Pi(H) &= \{A \in \Pi(\Sigma(H)) : A \precsim H\}, \\ \Pi^{\uparrow}(H) &= \sup_{\precsim} \Pi(H) \\ &= \bigsqcup\{A \in \Pi(\Sigma(H)) : A \precsim H\}. \end{aligned}$$

By completeness of $\Pi(\Sigma(H))$, if $\Pi(H)$ is nonempty, then $\Pi^{\uparrow}(H)$ is a process. The next proposition states that in this case $\Pi^{\uparrow}(H) \in \Pi(H)$.

Proposition 3.3 *For all* $A \in \Pi(\Sigma(H))$

$$A \precsim \Pi^{\uparrow}(H) \iff A \precsim H.$$

Proof. ($\Leftarrow$-part.) This part follows from the supremality of $\Pi^{\uparrow}(H)$.

($\Rightarrow$-part.) If $\Pi(H)$ is empty then $\Pi^{\uparrow}(H)$ is empty. No process $A \in \Pi(\Sigma(H))$ reduces $\Pi^{\uparrow}(H)$, so the implication is satisfied. If $\Pi(H)$ is nonempty then let $A \in \Pi(\Sigma(H))$ such that $A \precsim \Pi^{\uparrow}(H)$. From the definition of nondeterministic choice it can be derived that

$$\mathrm{L}(\Pi^{\uparrow}(H)) = \bigcup\{\mathrm{L}(A) : A \in \Pi(H)\},$$

and for all $s \in \mathrm{L}(\Pi^{\uparrow}(H))$

$$\mathrm{ref}(\Pi^{\uparrow}(H), s) = \bigcup\{\mathrm{ref}(A, s) : A \in \Pi(H) \wedge s \in \mathrm{L}(A)\}.$$

It follows that

$$\begin{aligned} s \in \mathrm{L}(A) &\Rightarrow s \in \mathrm{L}(\Pi^{\uparrow}(H)) \\ &\Rightarrow s \in \bigcup\{\mathrm{L}(A') : A' \in \Pi(H)\} \\ &\Rightarrow \exists A' \in \Pi(H) \text{ s.t. } s \in \mathrm{L}(A') \\ &\Rightarrow \qquad\qquad [\mathrm{L}(A') \subseteq \mathrm{L}(H)] \\ &\quad\; s \in \mathrm{L}(H). \end{aligned}$$

So $\mathrm{L}(A) \subseteq \mathrm{L}(H)$. Let $s \in \mathrm{L}(A)$. It follows that

$$\begin{aligned} R \in \mathrm{ref}(A, s) &\Rightarrow R \in \mathrm{ref}(\Pi^{\uparrow}(H), s) \\ &\Rightarrow R \in \bigcup\{\mathrm{ref}(A', s) : A' \in \Pi(H) \wedge s \in \mathrm{L}(A')\} \\ &\Rightarrow \exists A' \in \Pi(H) \text{ s.t. } s \in \mathrm{L}(A') \wedge R \in \mathrm{ref}(A', s) \\ &\Rightarrow \qquad\qquad [\mathrm{ref}(A', s) \subseteq \mathrm{ref}(H, s)] \\ &\quad\; R \in \mathrm{ref}(H, s). \end{aligned}$$

So $\mathrm{ref}(A, s) \subseteq \mathrm{ref}(H, s)$ for all $s \in \mathrm{L}(A)$. □

As $\Pi^{\uparrow}(H) \precsim \Pi^{\uparrow}(H)$ it follows that $\Pi^{\uparrow}(H) \precsim H$. So if $\Pi(H)$ is nonempty, then $\Pi^{\uparrow}(H) \in \Pi(H)$.

In Section 3.5 an algorithm is presented that computes process $\Pi^{\uparrow}(H)$. It removes states and refusals from the behavior state representation of H until conditions ii – v of Definition 2.5 are satisfied. Only states and refusals are removed that violate one of the conditions. If the resulting system is empty then no process in $\Pi(\Sigma)$ exists that reduces H. The algorithm is similar to algorithms that compute the supremal controllable sublanguage of a given language [20, 31, 42]. It will be shown that the algorithm has linear complexity in the size of the behavior state space of H.

This complexity depends crucially on the used behavior state representation. If processes G and E are given as ordinary automata, then they first need to be transformed to behavior state representations. This step has worst case exponential complexity. Therefore, the whole procedure will have exponential complexity. If systems are given by any other representation method, then the total complexity of the synthesis procedure depends on the complexity of the transformation to behavior state representations. Further research is needed to determine this complexity.

The supremal supervisor $S^{\uparrow}$ can be computed with the use of the algorithm for $\Pi^{\uparrow}$. For this let H be defined by

$$\begin{aligned}
\Sigma(H) &= \Sigma(G), \\
\mathrm{L}(H) &= \sup_{\subseteq}\{K \subseteq \Sigma^* : K = \bar{K} \text{ and } K \cap \mathrm{L}(G) \subseteq \mathrm{L}(E)\} \\
&= \{s \in \Sigma^* : \bar{s} \subseteq (\Sigma^* - \mathrm{L}(G)) \cup \mathrm{L}(E)\}, \\
\mathrm{ref}(H, s) &= \{R \subseteq \Sigma : \ \forall R_g \in \mathrm{ref}(G, s),\ R_g \cup R \in \mathrm{ref}(E, s) \text{ and} \\
&\qquad\qquad R \cap \Sigma_{uc} \subseteq \rho(\mathrm{L}(G), s) \qquad\qquad \}.
\end{aligned}$$

Theorem 3.4 *Let $S^{\uparrow}$ be the supremal solution of the basic supervisory control problem and let H be defined as above. Then $S^{\uparrow} = \Pi^{\uparrow}(H)$.*

Proof. By Theorem 3.2 and Proposition 3.3 it is sufficient to show that for any $A \in \Pi(\Sigma)$

$$A \precsim H \quad \Longleftrightarrow \quad G||A \precsim E \text{ and } A \text{ is complete.}$$

As A is a process, $\mathrm{L}(A)$ is prefix closed. So

$$\begin{aligned}
&\mathrm{L}(A) \subseteq \mathrm{L}(H) \\
&\quad \Longleftrightarrow \quad \mathrm{L}(A) \subseteq \sup_{\subseteq}\{K \subseteq \Sigma^* : K = \bar{K} \text{ and } K \cap \mathrm{L}(G) \subseteq \mathrm{L}(E)\} \\
&\quad \Longleftrightarrow \quad \mathrm{L}(A) \cap \mathrm{L}(G) \subseteq \mathrm{L}(E).
\end{aligned}$$

For all $s \in \mathrm{L}(A)$

$$\begin{aligned}
&\mathrm{ref}(A,s) \subseteq \mathrm{ref}(H,s)\\
&\iff \quad \forall R \in \mathrm{ref}(A,s),\ \forall R_{\mathrm{g}} \in \mathrm{ref}(G,s),\ R_{\mathrm{g}} \cup R \in \mathrm{ref}(E,s),\\
&\qquad\qquad \text{and } R \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s)\\
&\iff \quad \mathrm{ref}(G||A) \subseteq \mathrm{ref}(E,s), \text{ and}\\
&\qquad\quad A \text{ is complete.}
\end{aligned}$$

□

3.3. Least Restrictive Solutions

The supremal solution is not intended to be implemented directly. Often a supervisor is searched for that restricts the behavior of the controlled system as little as possible. Although $S^{\uparrow}$ is supremal, it is not optimal in this sense. Not only the language of $S^{\uparrow}$ is as large as possible, but also the refusal sets are maximal. This means that $S^{\uparrow}$ can refuse at least as much as any other legal supervisor. As the supervisor should restrict the behavior of the controlled system as little as possible, it is clear that $S^{\uparrow}$ is not a good candidate.

Definition 3.5 Supervisor $S^o \in \Pi(\Sigma)$, $S^o \precsim S^{\uparrow}$ is called *least restrictive* if for all $S \in \Pi(\Sigma)$

$$\begin{array}{lll}
S \precsim S^{\uparrow} & \Rightarrow \quad \mathrm{L}(G||S) \subseteq \mathrm{L}(G||S^o) & \text{and} \\
 & \quad\ \ \mathrm{ref}(G||S^o,s) \subseteq \mathrm{ref}(G||S,s) & \text{for all } s \in \mathrm{L}(G||S).
\end{array}$$

Note that the refusal sets are ordered in the opposite direction compared to the reduction relation.

Another reason why $S^{\uparrow}$ is not a good candidate to be implemented directly is that it is nondeterministic. By definition, $S^{\uparrow}$ can be considered as the nondeterministic choice between all supervisors that solve the supervisory control problem. It is usually not a good idea to implement the nondeterministic choice between two or more supervisors. It would require all alternatives to be built, but only one supervisor will be nondeterministically chosen and actually used. It is like building all alternatives, put them in a big sack, close your eyes, and pick one out of the sack which will then be used. So a supervisor which is intended to be implemented should be deterministic. Of course, nondeterministic supervisors are useful to describe, in an intermediate step, the set of all legal solutions. In subsequent steps this set can be narrowed down until only one deterministic supervisor remains. This deterministic supervisor can then be implemented.

Note that the situation is different in stochastic game theory [6]. In that field it has been shown that for some control problems nondeterministic (i.e. stochastic) controllers are optimal. In those control problems it is assumed that the controllers know the control law of their opponents. Nondeterministic controllers can be optimal in this situation because they introduce uncertainty in the control law. So, less information is revealed to the opponents. As we are considering only single supervisor control problems in this part of the thesis, these considerations are not relevant.

In general the language of the supremal supervisor contains traces that are not contained in G. These traces will never be executed in the controlled system. So it is not necessary that these traces are included in the language of the supervisor. This is the third reason why $S^{\uparrow}$ should not be implemented directly. Next we will derive a supervisor from $S^{\uparrow}$ that is least restrictive, deterministic, and the language of this supervisor is contained in $\mathrm{L}(G)$.

Lemma 3.6 *For all* $s \in \mathrm{L}(S^{\uparrow}) \cap \mathrm{L}(G)$, $\rho(\mathrm{L}(G), s) \in \mathrm{ref}(S^{\uparrow}, s)$.

Proof. We will construct a supervisor, whose refusal sets contain $\rho(\mathrm{L}(G), s)$ and all refusals of $S^{\uparrow}$. Next we will show that this supervisor is equal to $S^{\uparrow}$. Let supervisor S' be defined by

$$\begin{array}{lcl} \mathrm{L}(S') & = & \mathrm{L}(S^{\uparrow}), \\ \mathrm{ref}(S', s) & = & \{R_s \cup R_g : R_s \in \mathrm{ref}(S^{\uparrow}, s),\ R_g \subseteq \rho(\mathrm{L}(G), s)\}. \end{array}$$

First we need to show that S' is a process, i.e. that it satisfies the five points of Definition 2.5. S' satisfies points *i* and *ii* because $\mathrm{L}(S') = \mathrm{L}(S^{\uparrow})$. It satisfies point *iii* because $\emptyset \in \mathrm{ref}(S^{\uparrow}, s)$ and $\emptyset \subseteq \rho(\mathrm{L}(G), s)$. Point *iv* follows directly from the construction of S', and point *v* is satisfied because $\rho(\mathrm{L}(S'), s) = \rho(\mathrm{L}(S^{\uparrow}), s)$ and $\mathrm{ref}(S^{\uparrow}, s) \subseteq \mathrm{ref}(S', s)$. So $S' \in \Pi(\Sigma)$.

As $\mathrm{ref}(S^{\uparrow}, s) \subseteq \mathrm{ref}(S', s)$ it follows directly that $S^{\uparrow} \precsim S'$. It remains to prove that $S' \precsim S^{\uparrow}$. We will show that S' solves the basic supervisory control problem. Then by Theorem 3.2 it follows that $S' \precsim S^{\uparrow}$.

We need to show that $G||S' \precsim E$ and that S' is complete. The language part of $G||S' \precsim E$ follows from.

$$\mathrm{L}(G||S') \quad = \quad \mathrm{L}(G||S^{\uparrow}) \quad \subseteq \quad \mathrm{L}(E).$$

For the refusal part let $s \in \mathrm{L}(G||S') = \mathrm{L}(G||S^{\uparrow})$.

$$\begin{array}{ll} & R \in \mathrm{ref}(G||S', s) \\ \Rightarrow & \exists R' \in \mathrm{ref}(S', s), R_g \in \mathrm{ref}(G, s) \text{ s.t. } R = R' \cup R_g \\ \Rightarrow & \exists R_s \in \mathrm{ref}(S^{\uparrow}, s), R_g' \subseteq \rho(\mathrm{L}(G), s), R_g \in \mathrm{ref}(G, s) \text{ s.t.} \\ & R = R_s \cup R_g' \cup R_g \\ \Rightarrow & \qquad [G \text{ satisfies points } iv \text{ and } v \text{ of Definition 2.5, so} \\ & \qquad\qquad R_g'' = R_g' \cup R_g \in \mathrm{ref}(G, s)] \\ & \exists R_s \in \mathrm{ref}(S^{\uparrow}, s), R_g'' \in \mathrm{ref}(G, s) \text{ s.t. } R = R_s \cup R_g'' \\ \Rightarrow & R \in \mathrm{ref}(G||S^{\uparrow}, s) \\ \Rightarrow & R \in \mathrm{ref}(E, s). \end{array}$$

So $G||S' \precsim E$. For the proof of completeness let $s \in \mathrm{L}(G||S') = \mathrm{L}(G||S^{\uparrow})$.

$$\begin{aligned}
& R \in \mathrm{ref}(S', s) \\
& \quad \Rightarrow \quad \exists R_s \in \mathrm{ref}(S^{\uparrow}, s), R_g \subseteq \rho(\mathrm{L}(G), s) \text{ s.t. } R = R_s \cup R_g \\
& \quad \Rightarrow \quad \exists R_s \in \mathrm{ref}(S^{\uparrow}, s), R_g \subseteq \rho(\mathrm{L}(G), s) \text{ s.t.} \\
& \qquad\quad R \cap \Sigma_{\mathrm{uc}} = (R_s \cup R_g) \cap \Sigma_{\mathrm{uc}} = (R_s \cap \Sigma_{\mathrm{uc}}) \cup (R_g \cap \Sigma_{\mathrm{uc}}) \\
& \quad \Rightarrow \qquad\qquad\qquad\qquad\qquad [S^{\uparrow} \text{ is complete}, R_g \subseteq \rho(\mathrm{L}(G), s)] \\
& \qquad\quad R \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G), s).
\end{aligned}$$

So S' is complete. Hence $S' = S^{\uparrow}$ and $\rho(\mathrm{L}(G), s) \in \mathrm{ref}(S^{\uparrow}, s)$ for all $s \in \mathrm{L}(S^{\uparrow}) \cap \mathrm{L}(G)$. □

Theorem 3.7 *Let* $S^o \in \Pi(\Sigma)$.

$$\left.\begin{array}{l} S^o \precsim S^{\uparrow}, \\ S^o \textit{ is least restrictive}, \\ \mathrm{L}(S^o) \subseteq \mathrm{L}(G), \\ S^o \textit{ is deterministic} \end{array}\right\} \quad \Longleftrightarrow \quad S^o = \mathrm{Det}(\mathrm{L}(G||S^{\uparrow})).$$

Proof. ($\Rightarrow$-part) As S^o is least restrictive and $S^{\uparrow} \precsim S^{\uparrow}$ it follows from Definition 3.5 that $\mathrm{L}(G||S^{\uparrow}) \subseteq \mathrm{L}(G||S^o)$. By supremality of $S^{\uparrow}$

$$\begin{aligned} S^o \precsim S^{\uparrow} \quad & \Rightarrow \quad G||S^o \precsim G||S^{\uparrow} \\ & \Rightarrow \quad \mathrm{L}(G||S^o) \subseteq \mathrm{L}(G||S^{\uparrow}). \end{aligned}$$

As $\mathrm{L}(S^o) \subseteq \mathrm{L}(G)$ it follows that

$$\mathrm{L}(S^o) \quad = \quad \mathrm{L}(S^o) \cap \mathrm{L}(G) \quad = \quad \mathrm{L}(G||S^o) \quad = \quad \mathrm{L}(G||S^{\uparrow}).$$

As S^o is deterministic and a deterministic process is uniquely defined by its language, it follows that $S^o = \mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))$.

($\Leftarrow$-part) We need to prove that $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))$ satisfies the conditions in the proposition.

The condition $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})) \in \Pi(\Sigma)$ follows directly from Proposition 2.8.

The language of $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))$ satisfies

$$\mathrm{L}(\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))) \quad = \quad \mathrm{L}(G||S^{\uparrow}) \quad = \quad \mathrm{L}(S^{\uparrow}) \cap \mathrm{L}(G).$$

This proves the constraint $\mathrm{L}(\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})) \subseteq \mathrm{L}(G)$ and the language part of the constraint $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})) \precsim S^{\uparrow}$.

For the refusal part of this constraint let $s \in \mathrm{L}(G||S^{\uparrow})$.

$$\begin{aligned}
& R \in \mathrm{ref}(\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})), s) \\
& \quad \Rightarrow \quad R \subseteq \rho(\mathrm{L}(G||S^{\uparrow}), s) \\
& \quad \Rightarrow \quad R \subseteq \Sigma - \lambda(\mathrm{L}(G||S^{\uparrow}), s) = \Sigma - (\lambda(\mathrm{L}(G), s) \cap \lambda(\mathrm{L}(S^{\uparrow}), s)) \\
& \quad \Rightarrow \quad R \subseteq \rho(\mathrm{L}(G), s) \cup \rho(\mathrm{L}(S^{\uparrow}), s).
\end{aligned}$$

By Lemma 3.6 $\rho(\mathrm{L}(G), s) \in \mathrm{ref}(S^{\uparrow}, s)$. Then, by point *v* of Definition 2.5 $\rho(\mathrm{L}(G), s) \cup \rho(\mathrm{L}(S^{\uparrow}), s) \in \mathrm{ref}(S^{\uparrow}, s)$, and by point *iv* of the same definition $R \in \mathrm{ref}(S^{\uparrow}, s)$. Thus, for all $s \in \mathrm{L}(G||S^{\uparrow})$, $\mathrm{ref}(\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})), s) \subseteq \mathrm{ref}(S^{\uparrow}, s)$. And hence $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})) \sqsubseteq S^{\uparrow}$.

Finally we will show that $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))$ is least restrictive. Let $S \in \Pi(\Sigma)$ such that $S \sqsubseteq S^{\uparrow}$. Then

$$\begin{aligned} \mathrm{L}(G||S) &= \mathrm{L}(G) \cap \mathrm{L}(S) \\ &\subseteq \mathrm{L}(G) \cap \mathrm{L}(S^{\uparrow}) \\ &= \mathrm{L}(G||S^{\uparrow}) \\ &= \mathrm{L}(G||\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))). \end{aligned}$$

Also

$$\begin{aligned} &\mathrm{L}(G||S) \subseteq \mathrm{L}(G||S^{\uparrow}) \\ &\quad \Rightarrow \quad \lambda(\mathrm{L}(G||S), s) \subseteq \lambda(\mathrm{L}(G||S^{\uparrow}), s), \quad \forall s \in \mathrm{L}(G||S) \\ &\quad \Rightarrow \quad \rho(\mathrm{L}(G||S^{\uparrow}), s) \subseteq \rho(\mathrm{L}(G||S), s), \quad \forall s \in \mathrm{L}(G||S). \end{aligned}$$

Then, for all $s \in \mathrm{L}(G||S)$,

$$\begin{aligned} R \in \mathrm{ref}(\mathrm{Det}(\mathrm{L}(G||S^{\uparrow})), s) \quad &\Rightarrow \quad R \subseteq \rho(\mathrm{L}(G||S^{\uparrow}), s) \\ &\Rightarrow \quad R \subseteq \rho(\mathrm{L}(G||S), s) \\ &\Rightarrow \qquad\qquad [\text{Point } v \text{ of Definition 2.5}] \\ &\qquad\; R \in \mathrm{ref}(G||S, s). \end{aligned}$$

Hence $\mathrm{Det}(\mathrm{L}(G||S^{\uparrow}))$ is least restrictive. □

If $S^{\uparrow}$ is nonempty, then by Theorem 3.7 there always exists a deterministic supervisor. So if there exists a, possibly nondeterministic, supervisor then there also exists a deterministic solution.

Intuitively one might expect that a least restrictive supervisor is always deterministic, because a deterministic process refuses less than a nondeterministic process which generates the same language. This can be deduced from the points *iii* – *v* of Definition 2.5. For all $A \in \Pi(\Sigma)$

$$2^{\rho(\mathrm{L}(A), s)} \quad \subseteq \quad \mathrm{ref}(A, s).$$

So, for all $s \in \mathrm{L}(A)$

$$\mathrm{ref}(\mathrm{Det}(\mathrm{L}(A)), s) \quad = \quad 2^{\rho(\mathrm{L}(A), s)} \quad \subseteq \quad \mathrm{ref}(A, s).$$

The following example will show that in general a least restrictive supervisor does not have to be deterministic. Even if its language is contained in $\mathrm{L}(G)$.

$G = E = S = G||S$:

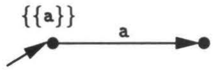


Figure 3.1: S is least restrictive.

Example 3.8 Let G, E and S be given as in Figure 3.1. Event a is controllable. It is not difficult to verify that $G||S = G = S$, that $G||S \underset{\sim}{\sqsubset} E$, that S is complete (so $S \underset{\sim}{\sqsubset} S^{\uparrow}$), and that $\mathrm{L}(S) \subseteq \mathrm{L}(G)$. From the definition of synchronous composition it can be deduced that for all $A \in \Pi(\Sigma)$, for all $s \in \mathrm{L}(G||A)$, $\mathrm{ref}(G, s) \subseteq \mathrm{ref}(G||A, s)$. As $G||S = G$ it follows that S is least restrictive. But S is not deterministic because it can initially execute event a as well as refuse it. When we compare S with the deterministic supervisor defined by $\mathrm{L}(S)$, we see that $G||S = G||\mathrm{Det}(\mathrm{L}(S))$. So S is not more restrictive than $\mathrm{Det}(\mathrm{L}(S))$. Informally, a user of the controlled system $G||S$ cannot detect whether event a is refused by G or by S.

3.4. Language Based Construction Method

Above, a construction scheme is presented to construct a least restrictive supervisor using the supremal supervisor. As this least restrictive supervisor is deterministic, it is defined by a language. This language can also be constructed in a more direct way, using language based methods.

Ramadge and Wonham showed that for the existence of a complete supervisor in a deterministic setting, the existence of a controllable language is a necessary and sufficient condition [52]. We will show that for nondeterministic systems the language also has to satisfy another condition, which is called reducibility. This reducibility condition guarantees that the supervisor refuses only events such that the refusal sets of the controlled system are contained in the refusal sets of the specification.

Definition 3.9 Let $G, E \in \Pi(\Sigma)$. Let K be a language contained in $\mathrm{L}(G)$ and $\mathrm{L}(E)$. K is *controllable* (w.r.t. G) if

$$K\Sigma_{\mathrm{uc}} \cap \mathrm{L}(G) = K.$$

K is *reducible* (w.r.t. G, E) if

$$\forall s \in K,\ \forall R_{\mathrm{g}} \in \mathrm{ref}(G, s),\ \rho(K, s) \cup R_{\mathrm{g}} \in \mathrm{ref}(E, s)\,.$$

An interpretation of the reducibility condition follows from the definition of reduction and synchronous composition. Observe that

$$\begin{aligned}&\mathrm{ref}(G||S,s) \subseteq \mathrm{ref}(E,s) && \Longleftrightarrow\\ &\forall R_{\mathrm{g}} \in \mathrm{ref}(G,s),\ \forall R_{\mathrm{s}} \in \mathrm{ref}(S,s),\ R_{\mathrm{g}} \cup R_{\mathrm{s}} \in \mathrm{ref}(E,s).\end{aligned}$$

If S is deterministic then $R_{\mathrm{s}} \in \mathrm{ref}(S,s)$ if and only if $R_{\mathrm{s}} \subseteq \rho(\mathrm{L}(S),s)$. As ref(E,s) is subset closed, it follows that

$$\begin{aligned}&\forall R_{\mathrm{g}} \in \mathrm{ref}(G,s),\ \forall R_{\mathrm{s}} \subseteq \rho(\mathrm{L}(S),s),\ R_{\mathrm{g}} \cup R_{\mathrm{s}} \in \mathrm{ref}(E,s) && \Longleftrightarrow\\ &\forall R_{\mathrm{g}} \in \mathrm{ref}(G,s),\ \rho(\mathrm{L}(S),s) \cup R_{\mathrm{g}} \in \mathrm{ref}(E,s).\end{aligned}$$

So, if S is deterministic then $\mathrm{ref}(G||S,s) \subseteq \mathrm{ref}(E,s)$ for all $s \in \mathrm{L}(G||S) \subseteq \mathrm{L}(E)$, if and only if $\mathrm{L}(S)$ is reducible.

The following lemma redefines the controllability condition in failure semantics terminology.

Lemma 3.10 *Let K be a prefix closed language contained in $\mathrm{L}(G)$. Then K is controllable if and only if*

$$\forall s \in K,\ \rho(K,s) \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s).$$

Proof.

$$\begin{aligned}&K\Sigma_{\mathrm{uc}} \cap \mathrm{L}(G) \subseteq K\\ &\quad\Longleftrightarrow\quad (s \in K \wedge \sigma \in \Sigma_{\mathrm{uc}} \wedge s\sigma \in \mathrm{L}(G)) \Rightarrow s\sigma \in K\\ &\quad\Longleftrightarrow\quad \neg(s \in K \wedge \sigma \in \Sigma_{\mathrm{uc}} \wedge s\sigma \in \mathrm{L}(G)) \vee s\sigma \in K\\ &\quad\Longleftrightarrow\quad \neg s \in K \vee \neg\sigma \in \Sigma_{\mathrm{uc}} \vee \neg s\sigma \in \mathrm{L}(G) \vee s\sigma \in K\\ &\quad\Longleftrightarrow\quad \neg s \in K \vee \neg\sigma \in \Sigma_{\mathrm{uc}} \vee \neg s\sigma \notin K \vee s\sigma \notin \mathrm{L}(G)\\ &\quad\Longleftrightarrow\quad (s \in K \wedge \sigma \in \Sigma_{\mathrm{uc}} \wedge s\sigma \notin K) \Rightarrow s\sigma \notin \mathrm{L}(G)\\ &\quad\Longleftrightarrow\quad \forall s \in K,\ \rho(K,s) \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s).\end{aligned}$$

□

Theorem 3.11 *Let $G, E \in \Pi(\Sigma)$. There exists a complete supervisor $S \in \Pi(\Sigma)$, such that $G||S \precsim E$ if and only if there exists a nonempty, prefix closed, controllable, and reducible language K contained in $\mathrm{L}(G)$ and $\mathrm{L}(E)$.*

Proof. (if part) Let K be a language satisfying the conditions of the theorem. It will be shown that $\mathrm{Det}(K)$ solves the control problem. As $K \subseteq \mathrm{L}(E)$ it follows that

$$\begin{aligned}\mathrm{L}(G||\mathrm{Det}(K)) &= \mathrm{L}(G) \cap K\\ &\subseteq \mathrm{L}(G) \cap \mathrm{L}(E)\\ &\subseteq \mathrm{L}(E).\end{aligned}$$

Let $R \in \mathrm{ref}(G||\mathrm{Det}(K),s)$. Then

$$\begin{array}{ll} \exists R_{\mathrm{g}} \in \mathrm{ref}(G,s),\ \exists R_{\mathrm{k}} \in \mathrm{ref}(\mathrm{Det}(K),s) \text{ s.t. } R = R_{\mathrm{g}} \cup R_{\mathrm{k}} & \\ \Rightarrow & [R_{\mathrm{k}} \subseteq \rho(K,s)] \\ \quad \exists R_{\mathrm{g}} \in \mathrm{ref}(G,s) \text{ s.t. } R \subseteq R_{\mathrm{g}} \cup \rho(K,s) & \\ \Rightarrow & [K \text{ is reducible and } \mathrm{ref}(E,s) \text{ is subset closed}] \\ \quad R \in \mathrm{ref}(E,s). & \end{array}$$

So $G||\mathrm{Det}(K) \precsim E$. As K is controllable it follows that

$$\begin{array}{lll} R \in \mathrm{ref}(\mathrm{Det}(K),s) & \Rightarrow & R \subseteq \rho(K,s) \\ & \Rightarrow & R \cap \Sigma_{\mathrm{uc}} \subseteq \rho(K,s) \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s). \end{array}$$

So $\mathrm{Det}(K)$ is complete.

(only if part) Let S be a complete supervisor such that $G||S \precsim E$. Let $K = \mathrm{L}(G||S)$. As $G||S$ is a process, K is nonempty and prefix closed. By the definition of synchronous composition $K \subseteq \mathrm{L}(G)$. As $G||S \precsim E$, $K \subseteq \mathrm{L}(E)$. It remains to prove that K is controllable and reducible. As $K = \mathrm{L}(G||S) = \mathrm{L}(G) \cap \mathrm{L}(S)$ it follows that

$$\begin{array}{ll} \lambda(K,s) = \lambda(\mathrm{L}(G),s) \cap \lambda(\mathrm{L}(S),s) & \\ \Rightarrow \quad \rho(K,s) = \rho(\mathrm{L}(G),s) \cup \rho(\mathrm{L}(S),s) & \\ \Rightarrow \quad \rho(K,s) \cap \Sigma_{\mathrm{uc}} = (\rho(\mathrm{L}(G),s) \cup \rho(\mathrm{L}(S),s)) \cap \Sigma_{\mathrm{uc}} & \\ \Rightarrow \quad \rho(K,s) \cap \Sigma_{\mathrm{uc}} = (\rho(\mathrm{L}(G),s) \cap \Sigma_{\mathrm{uc}}) \cup (\rho(\mathrm{L}(S),s) \cap \Sigma_{\mathrm{uc}}) & \\ \Rightarrow & [\rho(\mathrm{L}(S),s) \in \mathrm{ref}(S,s) \text{ and } S \text{ is complete}] \\ \quad \rho(K,s) \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s). & \end{array}$$

So K is controllable. Let $R_{\mathrm{g}} \in \mathrm{ref}(G,s)$. Then

$$R_{\mathrm{g}} \cup \rho(K,s) \quad = \quad R_{\mathrm{g}} \cup \rho(\mathrm{L}(G),s) \cup \rho(\mathrm{L}(S),s).$$

As $\rho(\mathrm{L}(S),s) \in \mathrm{ref}(S,s)$ and $R_{\mathrm{g}} \cup \rho(\mathrm{L}(G),s) \in \mathrm{ref}(G,s)$ it follows from the definition of synchronous composition that

$$R \cup \rho(K,s) \quad \in \quad \mathrm{ref}(G||S,s).$$

As $\mathrm{ref}(G||S,s) \subseteq \mathrm{ref}(E,s)$ it follows that $R \cup \rho(K,s) \in \mathrm{ref}(E,s)$. So K is reducible. □

It is not difficult to prove that the set of reducible languages contained in $\mathrm{L}(G) \cap \mathrm{L}(E)$ is closed under arbitrary unions, so a unique supremal element exists and is contained in the set. Let $K^{\uparrow}$ be this supremal. It can be efficiently computed (in the case of finite state systems) by an algorithm that removes states from the behavior state space of $G||E$. The algorithm is similar to algorithms that compute the supremal controllable sublanguage of a given language [20, 31, 42, 52, 62].

It can be shown that $K^{\uparrow}$ is equal to the language of the deterministic, least restrictive supervisor, with language contained in G. By the proof of Theorem 3.11 it follows that $G||\mathrm{Det}(K^{\uparrow}) \underset{\sim}{\sqsubseteq} E$. $\mathrm{Det}(K)$ is, of course, deterministic. By its construction $\mathrm{L}(\mathrm{Det}(K^{\uparrow})) = K^{\uparrow} \subseteq \mathrm{L}(G)$. And As $K^{\uparrow}$ is supremal, $\mathrm{Det}(K^{\uparrow})$ is least restrictive. So $K^{\uparrow}$ is equal to $\mathrm{L}(G||S^{\uparrow})$.

3.5. Algorithm to Compute $\Pi^{\uparrow}(H)$

This section contains the detailed algorithm to compute $\Pi^{\uparrow}(H)$ and a formal proof of its correctness. The algorithm removes states and refusals from the behavior state representation of H until conditions *ii* – *v* of Definition 2.5 are satisfied. It is assumed that the behavior state representation of H is given. This representation exists for any $(\Sigma, \mathrm{L}, \mathrm{ref})$ triple with a nonempty, prefix closed language. All $(\Sigma, \mathrm{L}, \mathrm{ref})$ triples used in this thesis have a nonempty, prefix closed language. If a triple is given with a language that is not prefix closed, then the $(\Sigma, \mathrm{L}, \mathrm{ref})$ triple restricted to the largest prefix closed sublanguage has to be considered. As $\Pi^{\uparrow}(H)$ will be prefix closed this restriction has no influence on the final result.

Define the following variables for all $q \in Q(H)$

$$\begin{aligned} \mathrm{status}(H,q) &\in \{\text{unchecked, ok, fault}\}, \\ \mathrm{rho}(H,q) &\in 2^{\Sigma(H)}, \\ \mathrm{parents}(H,q) &\in Q(H) \times \Sigma(H). \end{aligned}$$

Initiate for all $q \in Q(H)$

$$\begin{aligned} \mathrm{status}(H,q) &= \text{unchecked}, \\ \mathrm{rho}(H,q) &= \rho(H,q), \\ \mathrm{parents}(H,q) &= \{(q',\sigma) \in Q(H) \times \Sigma(H) : \delta(H,q',\sigma) = q\}. \end{aligned}$$

The algorithm is based on the algorithms presented in [20, 31]. Initially all states are labeled ‘unchecked’. Starting from the initial state all reachable states will be checked. This part is implemented by the procedure ‘down’. The procedure ‘local’ implements the actual test and the removal of refusals. If the test is successful then the state is marked ‘ok’. If the test fails then the state is marked ‘fault’. At the time when the parent states of a ‘fault’ state were checked, it was assumed that the ‘fault’ state would be ‘ok’. So, after this state is marked ‘fault’ the parent states need to be rechecked. This part is implemented by the procedure ‘up’. The algorithm is started with a call to ‘down$(H, Q_0(H))$’.

```
PROCEDURE down( $H$, $q \in Q(H)$ )
    IF status$(H,q)$ = unchecked
    THEN
        IF local$(H,q)$
```

```
            THEN
                FOR ALL $\sigma$ IN $\lambda(H,q)$ WHILE status$(H,q)$ = ok DO
                    down( $H, \delta(H,q,\sigma)$ )               (May change status$(H,q)$)
                ENDFOR
            ELSE
                FOR ALL $(q',\sigma)$ IN parents$(H,q)$ DO
                    up$(H,q')$
                ENDFOR
            ENDIF
        ENDIF
    END down

    PROCEDURE up( $H, q \in Q(H)$ )
        IF status$(H,q)$ = ok
        THEN
            IF NOT local$(H,q)$
            THEN
                FOR ALL $(q',\sigma')$ IN parents$(H,q)$ DO
                    up$(H,q')$
                ENDFOR
            ENDIF
        ENDIF
    END up

    FUNCTION local( $H$, $q \in Q(H)$ ) $\rightarrow$ Boolean
        $V := (2^{\Sigma(H)} - \text{ref}(H,q)) \cup \{R - \text{rho}(H,q) : R \in 2^{\Sigma(H)} - \text{ref}(H,q)\}$
        $W := \emptyset$
        WHILE $V - W \neq \emptyset$ DO
            LET $R \in V - W$
            $W := W \cup \{R\}$
            FOR ALL $\sigma \in \Sigma(H) - R$ DO
                $V := V \cup \{R \cup \{\sigma\}\}$
            ENDFOR
        ENDWHILE
        ref$(H,q)$ := ref$(H,q) - W$
        IF $\emptyset \in$ ref$(H,q)$
        THEN
            status$(H,q)$ := ok
            RETURN(true)
        ELSE
            status$(H,q)$ := fault
            FOR ALL $(q',\sigma')$ IN parents$(H,q)$ DO
                rho$(H,q')$ := rho$(H,q') \cup \{\sigma\}$
            ENDFOR
        RETURN(false)
    ENDIF
END local
```

Lemma 3.12 *If $Q(H)$ is finite, then the algorithms halts after a finite number of steps.*

Proof. First we will show that the function local stops after a finite number of steps. Note that refusals are never removed from the sets V and W. After each step in the while loop one refusal is added to W. As refusal sets are finite and $V \subseteq \text{ref}(X, q)$, eventually W will become equal to V after a finite number of steps. As also the number of parents of a state is finite, the for loop halts after a finite number of steps. So the function local halts after a finite number of steps.

Next we will show that the function local is called only a finite number of times. Procedure down calls local only for states that are unchecked. After the call the status of the state is not unchecked, and will never become unchecked again. So local is called at most once for every state from the procedure down. The procedure up calls local only for states with status ok. This happens at most once for every descendent that becomes fault. As the number of descendent is finite this happens only a finite number of times for each state. Hence, we can conclude that local is called only a finite number of times and therefore the algorithms halts after a finite number of steps. □

Let H_{alg} be the resulting automaton of the algorithm restricted to the states with status is ok.

During the execution the automaton H is changed only inside the function local. Let the sequence $H^0, H^1, \ldots, H^n$ be the subsequent forms of H. $H^0 = H$, $H^n = H_{\text{alg}}$, and for all $i \in 0 \ldots n$, H^i is the resulting automaton after the call local(H^{i-1}, q) for some $q \in Q(H)$.

Lemma 3.13 *Let $q \in Q(H)$. Let H^i and H^{i-1} be such that H^i is the resulting automaton after the call local(H^{i-1}, q). Let $R \subseteq 2^{\Sigma(H)}$.*

$$\forall R' \subseteq R,\ R' \in \text{ref}(H^{i-1}, q) \ \wedge\ R' \cup \mathit{rho}(H^{i-1}, q) \in \text{ref}(H^{i-1}, q)$$
$$\Longleftrightarrow$$
$$R \in \text{ref}(H^i, q).$$

Proof. Note that the function stops when $V - W = \emptyset$. Let V in the proof below denote V when local terminates. Then $V = W$.

($\Rightarrow$-part)

$$
\begin{aligned}
& R \notin \mathrm{ref}(H^i, q) \\
& \quad \Rightarrow \quad R \notin \mathrm{ref}(H^{i-1}, q) \ \vee \ R \in V \\
& \quad \Rightarrow \quad R \notin \mathrm{ref}(H^{i-1}, q) \ \vee \ R \cup \mathrm{rho}(H^{i-1}, q) \notin \mathrm{ref}(H^{i-1}, q) \ \vee \\
& \qquad\qquad \exists \sigma \in R \text{ s.t } R - \{\sigma\} \in V \\
& \quad \Rightarrow \quad R \notin \mathrm{ref}(H^{i-1}, q) \ \vee \ R \cup \mathrm{rho}(H^{i-1}, q) \notin \mathrm{ref}(H^{i-1}, q) \ \vee \\
& \qquad\qquad \exists \sigma \in R \text{ s.t } \ R - \{\sigma\} \notin \mathrm{ref}(H^{i-1}, q) \ \vee \\
& \qquad\qquad\qquad\qquad\quad (R - \{\sigma\}) \cup \mathrm{rho}(H^{i-1}, q) \notin \mathrm{ref}(H^{i-1}, q) \ \vee \\
& \qquad\qquad\qquad\qquad\quad \exists \sigma' \in R - \{\sigma\} \text{ s.t } R - \{\sigma, \sigma'\} \in V \\
& \quad \Rightarrow \quad \ldots \\
& \quad \Rightarrow \quad \exists R' \subseteq R \text{ s.t. } \ R' \notin \mathrm{ref}(H^{i-1}, q) \ \vee \\
& \qquad\qquad\qquad\qquad\qquad R' \cup \mathrm{rho}(H^{i-1}, q) \notin \mathrm{ref}(H^{i-1}, q).
\end{aligned}
$$

($\Leftarrow$-part)

$$
\begin{aligned}
& R \in \mathrm{ref}(H^i, q) \\
& \quad \Rightarrow \quad R \in \mathrm{ref}(H^{i-1}, q) \ \wedge \ R \notin V \\
& \quad \Rightarrow \quad R \in \mathrm{ref}(H^{i-1}, q) \ \wedge \ R \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q) \ \wedge \\
& \qquad\qquad \forall \sigma \in R, \ R - \{\sigma\} \notin V \\
& \quad \Rightarrow \quad R \in \mathrm{ref}(H^{i-1}, q) \ \wedge \ R \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q) \ \wedge \\
& \qquad\qquad \forall \sigma \in R, \ \ R - \{\sigma\} \in \mathrm{ref}(H^{i-1}, q) \ \wedge \\
& \qquad\qquad\qquad\qquad R - \{\sigma\} \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q) \ \wedge \\
& \qquad\qquad\qquad\qquad \forall \sigma' \in R - \{\sigma\}, \ R - \{\sigma, \sigma'\} \notin V \\
& \quad \Rightarrow \quad \ldots \\
& \quad \Rightarrow \quad \forall R' \subseteq R, \ \ R' \in \mathrm{ref}(H^{i-1}, q) \ \wedge \\
& \qquad\qquad\qquad\qquad R' \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q).
\end{aligned}
$$

□

Lemma 3.14 *Let $q \in Q(H)$. Let H^i and H^{i-1} be such that H^i is the resulting automaton after the call local(H^{i-1}, q). If status(H^i, q) = ok, then the refusal set $\mathrm{ref}(H^i, q)$ is the largest subset of $\mathrm{ref}(H^{i-1}, q)$ which satisfies*

$$\emptyset \in \mathrm{ref}(H^i, q),$$

and for all $R \in \mathrm{ref}(H^i, q)$

$$
\begin{aligned}
& \forall R' \subseteq R, \ R' \in \mathrm{ref}(H^i, q), \ \text{and} \\
& R \cup \mathit{rho}(H^i, q) \in \mathrm{ref}(H^i, q).
\end{aligned}
$$

Proof. First we will prove that $\mathrm{ref}(H^i, q)$ satisfies the given properties. As $\mathrm{status}(H^i, q) = \mathrm{ok}$ it follows directly from the function that $\emptyset \in \mathrm{ref}(H^i, q)$. Let $R \in \mathrm{ref}(H^i, q)$. Then

$$
\begin{array}{ll}
R' \subseteq R \ \wedge\ R \in \mathrm{ref}(H^i, q) & \\
\quad \Rightarrow & [\text{By Lemma 3.13}] \\
R' \subseteq R \ \wedge\ \forall R'' \subseteq R,\ \ R'' \in \mathrm{ref}(H^{i-1}, q) \wedge & \\
\qquad\qquad\qquad\qquad R'' \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q) & \\
\quad \Rightarrow & \\
\forall R'' \subseteq R',\ R'' \in \mathrm{ref}(H^{i-1}, q) \ \wedge\ R'' \cup \mathrm{rho}(H^{i-1}, q) \in \mathrm{ref}(H^{i-1}, q) & \\
\quad \Rightarrow & \\
R' \in \mathrm{ref}(H^i, q). &
\end{array}
$$

For the proof of the last line note that if $\mathrm{status}(H^i, q) = \mathrm{ok}$, then $\mathrm{rho}(H^i, q) = \mathrm{rho}(H^{i-1}, q)$. Let $\rho := \mathrm{rho}(H^i, q) = \mathrm{rho}(H^{i-1}, q)$.

$$
\begin{array}{ll}
R \cup \rho \notin \mathrm{ref}(H^i, q) & \\
\quad \Rightarrow & [\text{By Lemma 3.13}] \\
\exists R' \subseteq R \cup \rho \text{ s.t. } R' \notin \mathrm{ref}(H^{i-1}, q) \ \vee\ R' \cup \rho \notin \mathrm{ref}(H^{i-1}, q) & \\
\quad \Rightarrow & [\text{By initialization of } V] \\
R' - \rho = (R' \cup \rho) - \rho \in V \ \wedge\ R' - \rho \subseteq (R \cup \rho) - \rho \subseteq R & \\
\quad \Rightarrow & [\text{WHILE loop}] \\
R \in V & \\
\quad \Rightarrow & \\
R \notin \mathrm{ref}(H^i, q). &
\end{array}
$$

Hence $\mathrm{ref}(H^i, q)$ satisfies the properties.

Next we will prove that if $\mathrm{ref}^{\square} \subseteq \mathrm{ref}(H^{i-1}, q)$ satisfies the stated properties, then $\mathrm{ref}^{\square} \subseteq \mathrm{ref}(H^i, q)$. Let $\rho := \mathrm{rho}(H^i, q) = \mathrm{rho}(H^{i-1}, q)$.

$$
\begin{array}{lll}
R \in \mathrm{ref}^{\square} & \Rightarrow & \forall R' \subseteq R,\ R' \in \mathrm{ref}^{\square} \ \wedge\ R \cup \rho \in \mathrm{ref}^{\square} \\
& \Rightarrow & \hfill [\mathrm{ref}^{\square} \subseteq \mathrm{ref}(H^{i-1}, q)] \\
& & \forall R' \subseteq R,\ R' \in \mathrm{ref}(H^{i-1}, q) \ \wedge\ R' \cup \rho \in \mathrm{ref}(H^{i-1}, q) \\
& \Rightarrow & \hfill [\text{By Lemma 3.13}] \\
& & R \in \mathrm{ref}(H^i, q).
\end{array}
$$

So $\mathrm{ref}(H^i, q)$ is supremal. □

Theorem 3.15 $H_{\mathrm{alg}} = \Pi^{\uparrow}(H)$.

Proof.

($H_{\text{alg}} \precsim \Pi^{\uparrow}(H)$.) As $\Pi^{\uparrow}(H)$ is supremal it is sufficient to prove that H_{alg} is a process and that $H_{\text{alg}} \precsim H$.

As H_{alg} is defined by an automaton conditions *i* and *ii* of Definition 2.5 are automatically satisfied. All states of H_{alg} have status ok. Initially all states have status unchecked. So all states are checked at least once by the function local. Consider state $q \in Q(H)$. After a call of local(H,q), by Lemma 3.14, $\emptyset \in \text{ref}(H,q)$ and for all $R \in \text{ref}(H,q)$

$$\forall R' \subseteq R,\ R' \in \text{ref}(H,q), \text{ and}$$
$$R \cup \text{rho}(H,q) \in \text{ref}(H,q).$$

After the first call of local(H,q) the item rho(H,q) changes when one of the descendants of q becomes fault. If this happens then local(H,q) is called again via the procedure up. After this call the conditions are satisfied again. So it can be concluded that at the end of the algorithm all states satisfy the conditions. Note that if $\text{ref}(H,q) = \emptyset$ then the conditions are also satisfied. Note also that during the execution of the algorithm, at the end of each call to local, for all $q \in Q(H)$

$$\begin{aligned}\text{rho}(H,q) \quad = \quad \{\sigma \in \Sigma(H) : \ & \text{status}(H, \delta(H,q,\sigma)) = \text{fault} \vee \\ & \delta(H,q,\sigma) = \emptyset\}.\end{aligned}$$

So $\rho(\text{L}(H_{\text{alg}}), s) = \text{rho}(H_{\text{alg}}, [s]_H)$. And thus H_{alg} satisfies also condition *iii-v* of Definition 2.5. Hence H_{alg} is a process.

As H_{alg} is obtained from H by removing states and refusals, it follows automatically that $H_{\text{alg}} \precsim H$. It can be concluded that $H_{\text{alg}} \precsim \Pi^{\uparrow}(H)$.

($\Pi^{\uparrow}(H) \precsim H_{\text{alg}}$.) This part will be proven by complete induction. Consider the following induction hypothesis. For all states $[s]_H \in Q(H)$,

$$\text{status}(H^j, [s]_H) = \text{fault} \quad \Rightarrow \quad s \notin \text{L}(\Pi^{\uparrow}(H)),$$

and for all $s \in \text{L}(\Pi^{\uparrow}(H))$

$$\begin{aligned}\text{rho}(H^j, [s]_H) &\subseteq \rho(\text{L}(\Pi^{\uparrow}(H)), s), \\ \text{ref}(\Pi^{\uparrow}(H), s) &\subseteq \text{ref}(H^j, [s]_H).\end{aligned}$$

For the inductive step it is assumed that the induction hypothesis holds for all $j \in 0, \ldots, i-1$. It needs to be proven that the hypothesis also holds for $j = i$.

Assume $\text{status}(H^i, [s]_H) = \text{fault}$, but $s \in \text{L}(\Pi^{\uparrow}(H))$. Then by the negation of the first part of the induction hypothesis $\text{status}(H^{i-1}, [s]_H) \neq \text{fault}$. So it must be that local$(H^{i-1}, [s]_H)$ returns false. H^i is the resulting automaton after this call to local.

$$\begin{array}{ll}
\text{local}(H^{i-1}, [s]_H) = \text{ false} & \\
\quad\Rightarrow & \\
\emptyset \notin \text{ref}(H^i, [s]_H) & \\
\quad\Rightarrow & \text{[By Lemma 3.13]} \\
\emptyset \notin \text{ref}(H^{i-1}, [s]_H) \ \vee \ \text{rho}(H^{i-1}, [s]_H) \notin \text{ref}(H^{i-1}, [s]_H) & \\
\quad\Rightarrow & \text{[By the second part of the induction hypothesis} \\
 & \text{and the assumption that } s \in \text{L}(\Pi^{\uparrow}(H))] \\
\emptyset \notin \text{ref}(\Pi^{\uparrow}(H), s) \ \vee \ \rho(\text{L}(\Pi^{\uparrow}(H)), s) \notin \text{ref}(\Pi^{\uparrow}(H), s) & \\
\quad\Rightarrow & \text{[By points } iii \text{ and } v \text{ of Definition 2.5]} \\
s \notin \text{L}(\Pi^{\uparrow}(H)). &
\end{array}$$

But this contradicts our assumptions. We can conclude that if status$(H^i, [s]_H)$ is fault, then $s \notin \text{L}(\Pi^{\uparrow}(H))$.

Assume again that that local$(H^{i-1}, [s]_H)$ returns false. As ref$(H^i, [v]_H) =$ ref$(H^{i-1}, [v]_H)$ for all $[v]_H \neq [s]_H$, it follows that

$$\begin{array}{ll}
v \in \text{L}(\Pi^{\uparrow}(H)) & \\
\quad\Rightarrow & [v \in [s]_H \ \Rightarrow \ v \notin \text{L}(\Pi^{\uparrow}(H))] \\
v \notin [s]_H & \\
\quad\Rightarrow & \\
\text{ref}(\Pi^{\uparrow}(H), v) \subseteq \text{ref}(H^{i-1}, [v]_H) = \text{ref}(H^i, [v]_H). &
\end{array}$$

For all states $[v]_H$ that are not a parent of $[s]_H$, the rho-function remains unchanged. For all parents $([v]_H, \sigma)$ of state $[s]_H$ the rho-function rho$(H^i, [v]_H)$ is adapted such that rho$(H^i, [v]_H) =$ rho$(H^{i-1}, [v]_H) \cup \{\sigma\}$. As $v\sigma \in [s]_H$ implies that $v\sigma \notin \text{L}(\Pi^{\uparrow}(H))$ it follows that rho$(H^i, [v]_H) \subseteq \rho(\Pi^{\uparrow}(H), v)$.

Now assume that local$(H^{i-1}, [s]_H)$ returns true. Then for all states $[v]_H \in Q(H)$, rho$(H^i, [v]_H) =$ rho$(H^{i-1}, [v]_H)$. So for all $v \in \text{L}(\Pi^{\uparrow}(H))$, rho$(H^i, [v]_H) \subseteq \rho(\text{L}(\Pi^{\uparrow}(H)), v)$. It follows directly from Lemma 3.14 and the fact that $\Pi^{\uparrow}(H)$ is a process, that for all $v \in \text{L}(\Pi^{\uparrow}(H))$, ref$(\Pi^{\uparrow}(H), v) \subseteq$ ref$(H^i, [v]_H)$. This concludes the prove of the inductive step.

The initial step follows from the fact that at the beginning of the algorithm no state of H has status fault and that $\Pi^{\uparrow}(H) \precsim H$. So

$$\begin{array}{l}
\text{L}(\Pi^{\uparrow}(H) \subseteq \text{L}(H) \\
\quad\Rightarrow \quad \forall s \in \text{L}(\Pi^{\uparrow}(H)),\ \lambda(\Pi^{\uparrow}(H), s) \ \subseteq \ \lambda(\text{L}(H), s) \\
\quad\Rightarrow \quad \forall s \in \text{L}(\Pi^{\uparrow}(H)),\ \text{rho}(H, [s]_H) \ = \rho(\text{L}(H), s) \ \subseteq \ \rho(\Pi^{\uparrow}(H)),
\end{array}$$

and

$$\forall s \in \text{L}(\Pi^{\uparrow}(H)),\ \text{ref}(\Pi^{\uparrow}(H), s) \subseteq \text{ref}(H, [s]_H).$$

We have proven that the induction hypothesis holds for all $H^i, i \in 0 \ldots n$, so also for $H_{\mathbf{alg}}$. Hence it can be concluded that $\Pi^{\uparrow}(H) \precsim H_{\mathbf{alg}}$. □

The function local is called once initially in every node and then once for every of the $|\Sigma|$ descendants that becomes fault. The function local has to check every refusal. So it has a complexity in the order of $2^{|\Sigma|}$. The total algorithm has therefore a complexity of order $|Q(H)| \times (1 + |\Sigma|) \times 2^{|\Sigma|}$.

Chapter 4
Partial Specifications

One aspect of a specification is that it should be implementation independent. That is, a specification should describe *what* a system should do, not *how* it should be done. This has the advantage that two systems with a completely different implementation, but with the same specification are interchangeable. Consider, for instance, a car. All cars have a similar specification. They have a gas-pedal on the right, a brake in the middle, and optionally a clutch on the left. It is not necessary to know the implementation dependent aspects of the car, such as the number of pistons, or the way the fuel is injected. Just a specification given in events relevant for the user is sufficient to drive any car, from a family sedan to a high powered sportscar. A specification which does not contain all implementation details will be called a partial specification.

Partial specifications are also well suited for design problems in layered architectures, such as the ISO-OSI network model discussed in Chapter 1. Protocol design problems in layered architectures can be treated as control problems by considering the lower level service as uncontrolled system, the protocol as supervisor, and the higher level service as specification. Usually the lower level uses implementation events that are not used in the higher level. This leads very naturally to a control problem with partial specification.

Consider for instance the control problem to establish reliable transmission over an unreliable channel. This is the control problem for which the well known alternating bit protocol is a solution [13, 55, 57]. The lower level or uncontrolled system describes the behavior of the unreliable channel. It describes the transmission of messages and the transmission of control information needed by the protocol. It also describes the possibility that messages may get lost. The protocol or supervisor describes which messages have to be sent and at what time. It also describes when acknowledgements should be sent and when messages have to be retransmitted. From the outside the implementation should behave as a reliable channel. A message which is delivered to one side of the channel should eventually be received by the other side. This is described by the specification or higher level service. This description does not include the implementation details such as acknowledgements and the transmission of control information. These details are not relevant for a user of the system. The specification describes what the system does. In this case reliable transmission. It does not describe how this is done.

In this chapter the supervisory control problem with partial specification is formulated and it will be shown that it can be reduced to the basic supervisory control problem.

4.1. Projection

In order to be able to investigate the external or higher level description of a system we need a method to project internal events out of a description.

Definition 4.1 Let $\Sigma_e \subseteq \Sigma$ denote the set of *external events* and $\Sigma_i = \Sigma - \Sigma_e$ the set of *internal events*. Define p_e as the natural *projection* from traces in Σ^* to traces in Σ_e^*;

$$\begin{aligned} p_e(\varepsilon) &= \varepsilon, \\ p_e(s\sigma) &= \begin{cases} p_e(s)\sigma, & \text{if } \sigma \in \Sigma_e, \\ p_e(s), & \text{if } \sigma \notin \Sigma_e. \end{cases} \end{aligned}$$

The projection of a language is the projection of all its traces.

$$p_e(K) = \{p_e(s) : s \in K\}.$$

The inverse projection of p_e is defined to be

$$p_e^{-1}(s_e) = \{s \in \Sigma : p_e(s) = s_e\}.$$

Note that $p_e^{-1}(s)$ is a set of traces, i.e. a language. Similar, the inverse projection of a language is a set of languages.

$$p_e^{-1}(K_e) = \{K \subseteq \Sigma^* : p_e(K) = K_e\}.$$

Define

$$p_e^{\uparrow}(K_e) = \sup_{\subseteq} p_e^{-1}(K_e) = \{s \in \Sigma^* : p_e(s) \in K_e\}.$$

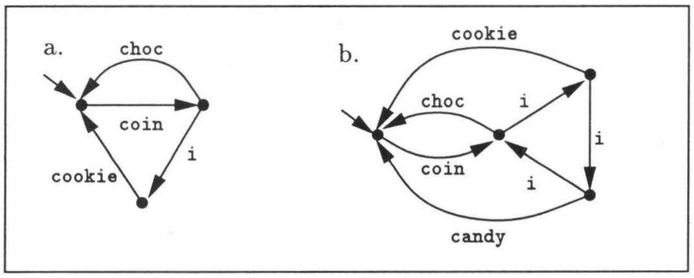


Figure 4.1: Internal events and divergence.

In the sequel a small p will be used to denote projections of traces, languages, and refusal sets. A large P will denote projections of processes (See Definition 4.3). A subscript indicates the event set on which is projected.

Consider a vending machine as shown in Figure 4.1.a. After inserting a coin, a chocolate bar can be obtained, provided the machine does not execute internal event i, after which only a cookie can be obtained. The question is what the projected system can refuse after a coin is inserted. Suppose a customer insists on having a cookie. He refuses to accept a chocolate bar. The machine must now execute the internal event because that is the only possible event that is not blocked. The system ends up in a state in which it must hand out a cookie. So, it is clear that the machine cannot refuse to engage in a cookie event. The machine can refuse to hand out a chocolate bar because the customer cannot prevent the internal event from occurring. After the internal event the vending machine cannot engage in the choc event.

If a system refuses external event set $R_e \subseteq \Sigma_e$, but it can still execute an internal event, then it may end up in a state in which it does not refuse this external event set. On the other hand, if the system does not refuse R_e, but it can still execute an internal event, then it may end up in a state in which it can refuse R_e. It turns out that the refusals of the projected system are defined by those states in which the machine cannot execute internal events. These states correspond with refusals that contain the set of internal events. The projection of the refusal set ref(A, s) on events set Σ_e is defined by

$$\mathrm{p_e}(\mathrm{ref}(A,s)) = \{R \subseteq \Sigma_e : R \cup \Sigma_i \in \mathrm{ref}(A,s)\}.$$

Divergence

But there is another problem. It may happen that a machine can execute internal events forever. See for example Figure 4.1.b. After a coin is inserted the machine can always choose to execute an internal event, because it cannot be blocked from the outside. To the customer it appears as if the machine refuses all external events. This phenomenon is called divergence [23].

A trace s is called divergent with respect to a system and a external event set, if the system can execute an unbounded number of internal events after s. We cannot write "an infinite number of events", because only finite traces are considered.

Definition 4.2 The set of *divergent traces* with respect to a language $K \subseteq \Sigma^*$ and external event set Σ_e is denoted by $\mathrm{div}(K, \Sigma_e)$, and defined by

$$\mathrm{div}(K, \Sigma_e) = \{s \in K : \forall n \in \mathbb{N},\ \exists s_i \in \Sigma_i^* \text{ s.t. } |s_i| > n \text{ and } ss_i \in K\}.$$

Let $A \in \Pi(\Sigma)$ be a process. For notational convenience we will use $\mathrm{div}(A, \Sigma_e)$ to denote $\mathrm{div}(\mathrm{L}(A), \Sigma_e)$.

As $\mathrm{div}(A, \Sigma_e)$ is a set of traces, i.e. a language, the projection $\mathrm{p_e}(\mathrm{div}(A, \Sigma_e))$ is well defined.

Definition 4.3 The *projection* of process $A \in \Pi(\Sigma)$ on alphabet Σ_e is the process $\mathrm{P_e}(A) \in \Pi(\Sigma_e)$, where

$$\begin{aligned}
\Sigma(\mathrm{P_e}(A)) &= \Sigma_e, \\
\mathrm{L}(\mathrm{P_e}(A)) &= \mathrm{p_e}(\mathrm{L}(A)), \\
\mathrm{ref}(\mathrm{P_e}(A), s_e) &= \begin{cases} 2^{\Sigma_e}, & \text{if } s_e \in \mathrm{p_e}(\mathrm{div}(A, \Sigma_e)), \\ \bigcup_{s \in \mathrm{p_e^{-1}}(s_e) \cap \mathrm{L}(A)} \mathrm{p_e}(\mathrm{ref}(A, s)), & \text{otherwise.} \end{cases}
\end{aligned}$$

In the definition above it is assumed that if a system can diverge then it has the ability to refuse all external events. In some situations this is a rather pessimistic assumption. Sometimes a more optimistic approach is justified. Consider for instance a network where a lost message causes automatically the retransmission of the message. The internal events 'message-lost' and 'retransmit' form together a loop of internal events. After retransmission the retransmitted message may also get lost, which causes the next retransmission. With a pessimistic point of view one can argue that the system can execute internal events indefinitely long, and can therefore refuse all external events. But usually it is assumed that eventually, after sufficient retransmissions, the network will be able to deliver the message. This can be interpreted as if the system cannot refuse the external 'message-received' event. It would go beyond the scope of this thesis to further investigate the consequences of this more optimistic interpretation of divergence [21]. In the sequel the pessimistic approach towards divergence will be used.

The next proposition follows directly from [11, Theorem 1].

Proposition 4.4 *Let $\Sigma_e \subseteq \Sigma$. Let A and $B \in \Pi(\Sigma)$ such that $\Sigma_e \subseteq \Sigma(A) = \Sigma(B)$.*

$$A \precsim B \quad \Rightarrow \quad \mathrm{P_e}(A) \precsim \mathrm{P_e}(B).$$

4.2. Supervisory Control Problem with Partial Specification

Given the definition of projection, we can formulate the supervisory control problem with partial specification.

The event set Σ will be divided into two subsets Σ_e and Σ_i. The external events (Σ_e) are those events that are relevant for the users of the system. The specification should be stated in terms of these events. The internal or implementation events (Σ_i) are not provided to the environment. They do not appear in the specification. They are, however, observable by the supervisor, because the supervisor is part of the implementation. The supervisory control problem with partial specification can be defined as follows.

Definition 4.5 Let $E \in \Pi(\Sigma_e)$ be the specification process, and let $G \in \Pi(\Sigma)$ be the uncontrolled system. Let $\Sigma_e \subseteq \Sigma$. The *supervisory control problem with partial specification* is to find a complete supervisor $S \in \Pi(\Sigma)$, such that $\mathrm{P}_e(G||S) \precsim E$.

In a language semantics setting the control problem with partial specification can be easily rewritten into a control problem with full specification. This is possible because the following relation holds.

$$\mathrm{p}_e(\mathrm{L}(G||S)) \subseteq \mathrm{L}(E) \iff \mathrm{L}(G||S) \subseteq \sup_{\subseteq} \mathrm{p}_e^{-1}(\mathrm{L}(E)). \tag{4.1}$$

Note that

$$\begin{aligned}
\sup_{\subseteq} \mathrm{p}_e^{-1}(\mathrm{L}(E)) &= \{s \in \Sigma^* : \mathrm{p}_e(s) \in \mathrm{L}(E)\} \\
&= \sup_{\subseteq} \{K \subseteq \Sigma^* : \mathrm{p}_e(K) \subseteq \mathrm{L}(E)\} \\
&= \bigcup \{K \subseteq \Sigma^* : \mathrm{p}_e(K) \subseteq \mathrm{L}(E)\}.
\end{aligned}$$

So $\sup_{\subseteq} \mathrm{p}_e^{-1}(\mathrm{L}(E))$ is equal to the union of all legal implementations, i.e. the union of all languages that are allowed as language of the controlled system.

The supremal solution, $S^{\uparrow}$, of the control problem with full specification (the right hand part of (4.1)) can be obtained using standard supervisory synthesis methods. It is also the supremal solution of the control problem with partial specification because

$$\mathrm{L}(S) \subseteq \mathrm{L}(S^{\uparrow}) \Rightarrow \mathrm{p}_e(\mathrm{L}(S)) \subseteq \mathrm{p}_e(L(S^{\uparrow})).$$

Also the control problem with partial specification where the inclusion relation is replaced by equality can be easily solved in a language semantics setting. Note that

$$\mathrm{p}_e(\mathrm{L}(G||S)) = \mathrm{L}(E) \Rightarrow \mathrm{L}(G||S) \subseteq \sup_{\subseteq} \mathrm{p}_e^{-1}(\mathrm{L}(E)). \tag{4.2}$$

Let $S^{\uparrow}$ be the supremal solution of the control problem with full specification. By (4.1) it holds that $p_e(\mathrm{L}(G||S^{\uparrow})) \subseteq \mathrm{L}(E)$. If $\mathrm{p}_e(\mathrm{L}(G||S^{\uparrow})) = \mathrm{L}(E)$ then a solution to the control problem is found. If $\mathrm{p}_e(\mathrm{L}(G||S^{\uparrow})) \subsetneq \mathrm{L}(E)$ then, as $S^{\uparrow}$ is supremal, there does not exists a solution to the control problem with partial specification and equality. To show this last point, assume that there exists an S which solves the control problem with partial specification and equality. Then by (4.2) it solves the control problem with full specification of which $S^{\uparrow}$ is the supremal solution. So $\mathrm{L}(S) \subseteq \mathrm{L}(S^{\uparrow}$. But then it also holds that

$$\mathrm{p_e}(\mathrm{L}(G||S)) \subseteq \mathrm{p_e}(\mathrm{L}(G||S)) \subsetneq \mathrm{L}(E).$$

This contradicts the assumption that S solves the control problem with partial specification and equality.

Let us try to apply the ideas mentioned above to nondeterministic systems. Define $\mathrm{P_e^{-1}}(E)$ as the set of all systems that project onto E.

$$\mathrm{P_e^{-1}}(E) = \{A \in \Pi(\Sigma) : \mathrm{P_e}(A) = E\}.$$

In Chapter 3 it was stated that $\Pi(\Sigma)$ forms a complete upper semi-lattice, with the reduction relation as partial ordering. Any subset of processes has a supremal, which is an element of $\Pi(\Sigma)$. However, this supremal does not necessarily have to be an element of the subset. The following example shows that in general

$$G||S \mathrel{\underset{\sim}{\sqsubseteq}} \sup_{\underset{\sim}{\sqsubseteq}} \mathrm{P_e^{-1}}(E) \quad \not\Rightarrow \quad \mathrm{P_e}(G||S) \mathrel{\underset{\sim}{\sqsubseteq}} E.$$

So in general it is not guaranteed that $\sup_{\underset{\sim}{\sqsubseteq}} \mathrm{P_e^{-1}}(E)$ is an element of $\mathrm{P_e^{-1}}(E)$.

Example 4.6 Let $E \in \Pi(\Sigma_e)$ be a process such that the refusal set after $\mathrm{p_e}(s)$ does not contain the complete external event set Σ_e. If we compute the inverse projection of E then this will include systems that can execute $s\Sigma_i^n$, with n some constant. Systems that allow $s\Sigma_i^*$ can diverge after s. When projected, they can refuse the whole external events set. This is not allowed by E. So systems that allow $s\Sigma_i^*$ are not an element of $\mathrm{P_e^{-1}}(E)$. The supremal element of $\{s\Sigma_i^n : n \in \mathbb{N}\}$ is $s\Sigma_i^*$. So the supremal element of $\mathrm{P_e^{-1}}(E)$ will allow $s\Sigma_i^*$. We have that the supremal element of $\mathrm{P_e^{-1}}(E)$ does not project onto E. Therefore $G||S \mathrel{\underset{\sim}{\sqsubseteq}} \sup_{\underset{\sim}{\sqsubseteq}} \mathrm{P_e^{-1}}(E)$ does not imply $\mathrm{P_e}(G||S) \mathrel{\underset{\sim}{\sqsubseteq}} E$.

4.3. Bounded Recurrence

In Example 4.6 we can see that if the number of internal events is bounded, then the system cannot diverge. The idea is now to consider only solutions that allow a bounded number of internal events. However, the difficulty is to find a good bound. If the bound is to strict then this may limit the behavior of the implementation to much. Consider the following points

- If G cannot diverge after trace $s \in \mathrm{L}(G||S)$ then neither can $G||S$, because $\mathrm{L}(G||S) \subseteq \mathrm{L}(G)$. So if $s \notin \mathrm{div}(G, \Sigma_e)$, then the number of internal events that $G||S$ can execute after trace s is bounded.

- Let G have a finite state space. If G can diverge after trace $s \in \mathrm{L}(G)$, then G can execute an unbounded number of internal events after trace s. It can execute more internal events than the state space of G has states. So it can execute a trace of internal events such that certain states of G are visited more than once by this trace. The trace will make a loop (cycle).
- The behavior of $G||S$ is related to the reachability of certain states. If, for instance, event b can be executed only in state q, and state q is not reachable, then event b can never be executed.

The idea is not to limit the number of internal events directly, but to limit the number of loops that sequences of internal events can make. In this way it is guaranteed that all states in the loop are still reachable. It will be shown that if the number of loops is bounded by a constant ≥ 2, then the external behavior of the implementation will not be restricted by this.

First, some definitions will be given to define processes that make at most two loops of internal events. In the next section, it will be shown how the control problem with partial specification can be reduced to the basic control problem. After that, an example will be given to illustrate the followed approach. It will also be shown in this example why at least two internal loops are needed to guarantee that the external behavior is not restricted by the followed approach.

Because the external behavior is relevant for the user of the system, we want to make it as least restrictive as possible. The internal behavior is invisible for the user. It is only relevant for the implementation. There is no reason why this behavior should be least restrictive. In fact it is even desirable to make the internal behavior as small as possible, in order to keep the implementation costs as small as possible [43]. In this thesis only implementations will be considered that make at most 2 loops of internal events when the specification cannot refuse the whole external event set. This is formalized by introducing the notion of bounded recurrence.

Definition 4.7 It will be said that trace $s' \in \Sigma^*$ is in the *last internal part* of trace $s \in \Sigma^*$ if $s' \in \overline{s}$ and $\mathrm{p_e}(s') = \mathrm{p_e}(s)$. Thus, if s' is in the last internal part of s then there exists an $s_\mathrm{i} \in \Sigma_\mathrm{i}^*$ such that $s's_\mathrm{i} = s$. Traces s' and s are equal up to some internal events at the end of trace s.

Let the *recurrence index* of trace $s \in \Sigma^*$ indicate how often the behavior state $[s]_G$ is visited by the last internal part of trace s.

$$\mathrm{r_i}(G, s) \quad = \quad |\{s' \in \overline{s} : \mathrm{p_e}(s) = \mathrm{p_e}(s') \wedge [s]_G = [s]'_G\}|.$$

Consider the behavior state representation of system G given in Example 4.18. The recurrence indices of the traces `a`, `aij`, and `aijij` are 1, 2, and 3 respectively. Note that, as $[s]_G$ is well defined for $s \notin \mathrm{L}(G)$, also $\mathrm{r_i}(G, s)$ is well defined for $s \notin \mathrm{L}(G)$.

The following lemma relates the divergent traces of process A to the recurrence index with respect to G.

Lemma 4.8 *Let $G \in \Pi(\Sigma)$ be a process with finite state space. Let $A \in \Pi(\Sigma)$ and let $s \in \mathrm{L}(A)$. Then*

$$s \in \mathrm{div}(A, \Sigma_e) \quad \Rightarrow \quad \exists s_i \in \Sigma_i^* \text{ s.t. } ss_i \in \mathrm{L}(A) \text{ and } \mathrm{r_i}(G, ss_i) > 2$$

Proof. Let $n \in \mathbb{N}$ be such that $2 \times |Q^+(G)| < n$. Let $s \in \mathrm{div}(A, \Sigma_e)$. Then, by the definition of divergent traces, there exists a trace $s_i \in \Sigma_i$ such that $|s_i| > n$ and $ss_i \in \mathrm{L}(A)$. As $2 \times |Q^+(G)| < |s_i|$ there must exist a state in $Q^+(G)$ which is visited at least three times by trace s_i. That is, there exist three distinct traces $v_i, w_i, z_i \in \overline{s_i}$ such that $[sv_i]_G = [sw_i]_G = [sz_i]_G$. Assume trace z_i is the longest, then $v_i \in \overline{z_i}$ and $w_i \in \overline{z_i}$. By the definition of recurrence index it follows that $\mathrm{r_i}(G, sz_i) > 2$. Note that $sz_i \in \mathrm{L}(A)$, because $sz_i \in \overline{ss_i} \subseteq \mathrm{L}(A)$. Note also that it is not required that any of the traces is an element of $\mathrm{L}(G)$. □

Definition 4.9 Trace $s \in \Sigma^*$ is called *bounded recurrent* (w.r.t G and E) if

$$\Sigma_e \notin \mathrm{ref}(E, \mathrm{p_e}(s)) \quad \Rightarrow \quad \mathrm{r_i}(G, s) \leq 2.$$

A language $K \subseteq \Sigma^*$ is called *bounded recurrent* if all traces $s \in K$ are bounded recurrent. Process $A \in \Pi(\Sigma)$ is called *bounded recurrent* if $\mathrm{L}(A)$ is bounded recurrent.

Note that if $s \notin \mathrm{div}(G, \Sigma_e)$ then G cannot make a loop of internal events after s. So, for all $s_i \in \Sigma_i^+$ such that $ss_i \in \mathrm{L}(G)$ we have that $[s]_G \neq [ss_i]_G$. Thus $\mathrm{r_i}(G, ss_i) = 1$ and hence ss_i is bounded recurrent. Note also that if $\Sigma_e \in \mathrm{ref}(E, \mathrm{p_e}(s))$ then s is defined to be bounded recurrent.

Lemma 4.10 *Let G and $A \in \Pi(\Sigma)$. Let G have a finite state space and let A be bounded recurrent. Then for all $s_e \in \mathrm{p_e}(\mathrm{L}(A))$,*

$$s_e \in \mathrm{p_e}(\mathrm{div}(A, \Sigma_e)) \quad \Rightarrow \quad \Sigma_e \in \mathrm{ref}(E, s_e).$$

Proof. Let $s_e \in \mathrm{p_e}(\mathrm{L}(A))$. Then

$$\begin{aligned}
& s_e \in \mathrm{p_e}(\mathrm{div}(A, \Sigma_e)) \\
& \quad \Rightarrow \quad \exists s \in \mathrm{p_e^{-1}}(s_e) \cap \mathrm{L}(A) \text{ s.t. } s \in \mathrm{div}(A, \Sigma_e) \\
& \quad \Rightarrow \quad \hfill [\text{By Lemma 4.8}] \\
& \qquad \exists s \in \mathrm{p_e^{-1}}(s_e) \cap \mathrm{L}(A),\ \exists s_i \in \Sigma_i^* \text{ s.t. } ss_i \in \mathrm{L}(A) \text{ and } \mathrm{r_i}(G, ss_i) > 2 \\
& \quad \Rightarrow \quad \hfill [A \text{ is bounded recurrent, } \mathrm{p_e}(ss_i) = s_e] \\
& \qquad \Sigma_e \in \mathrm{ref}(E, s_e).
\end{aligned}$$

□

4.4. Reduction of the Control Problem

In this section it will be shown that, if we restrict the set of implementations to bounded recurrent processes, then the control problem with partial specification can be reduced to the basic supervisory control problem. It will also be shown that this restriction does not limit the external behavior of the controlled system.

Definition 4.11 $E^{\uparrow} = \sup_{\underset{\sim}{\sqsubset}} \mathrm{P}_{\mathrm{e}}^{-1}(E) = \sup_{\underset{\sim}{\sqsubset}} \{A \in \Pi(\Sigma) : \mathrm{P}_{\mathrm{e}}(A) \underset{\sim}{\sqsubset} E\}$.

Proposition 4.12 *The process* $E^{\uparrow}$ *satisfies:*

$$\begin{aligned} \Sigma(E^{\uparrow}) &= \Sigma, \\ \mathrm{L}(E^{\uparrow}) &= \mathrm{p}_{\mathrm{e}}^{-\uparrow}(\mathrm{L}(E)) = \{s \in \Sigma^* : \mathrm{p}_{\mathrm{e}}(s) \in \mathrm{L}(E)\}, \\ \mathrm{ref}(E^{\uparrow}, s) &= \{R \subseteq \Sigma : \Sigma_{\mathrm{i}} \subseteq R \Rightarrow \Sigma_{\mathrm{e}} \cap R \in \mathrm{ref}(E, \mathrm{p}_{\mathrm{e}}(s))\}. \end{aligned}$$

This characterization can be used to construct $E^{\uparrow}$.

Proof. Let E_{con} be the process defined by the expressions in the proposition. We have to prove that $E^{\uparrow} = E_{\mathrm{con}}$. For all $n \in \mathbb{N}$ let A_n be the process defined by

$$\begin{aligned} \Sigma(A_n) &= \Sigma, \\ \mathrm{L}(A_n) &= \{s \in \Sigma^* : s \in \overline{\Sigma_{\mathrm{i}}^n}\sigma_1\overline{\Sigma_{\mathrm{i}}^n}\sigma_2 \ldots \sigma_m\overline{\Sigma_{\mathrm{i}}^n}, \sigma_1\sigma_2\ldots\sigma_m \in \mathrm{L}(E)\}, \\ \mathrm{ref}(A_n, s) &= \mathrm{ref}(E_{\mathrm{con}}, s). \end{aligned}$$

The construction of $\mathrm{L}(A)$ is such that any trace in $\mathrm{p}_{\mathrm{e}}^{-\uparrow}(\mathrm{L}(E))$, which has at most n internal events between every two external events is an element of $\mathrm{L}(A)$.

First, it will be proven that for all $n \in \mathbb{N}$, $\mathrm{P}_{\mathrm{e}}(A_n) \underset{\sim}{\sqsubset} E$. The language part of the reduction relation follows from $\mathrm{L}(\mathrm{P}_{\mathrm{e}}(A_n)) = \mathrm{p}_{\mathrm{e}}(\mathrm{L}(A_n)) = \mathrm{L}(E)$. Note that after any trace in $\mathrm{L}(A_n)$ only a bounded number of internal events are possible, so $\mathrm{div}(A_n, \Sigma_{\mathrm{e}}) = \emptyset$. The refusal part of $\mathrm{P}_{\mathrm{e}}(A_n)$ follows from

$$\begin{aligned} & \mathrm{ref}(\mathrm{P}_{\mathrm{e}}(A_n), s_{\mathrm{e}}) \\ = & \bigcup_{s \in \mathrm{p}_{\mathrm{e}}^{-1}(s_{\mathrm{e}}) \cap \mathrm{L}(A_n)} \mathrm{p}_{\mathrm{e}}(\mathrm{ref}(A_n, s)) \\ = & \bigcup_{s \in \mathrm{p}_{\mathrm{e}}^{-1}(s_{\mathrm{e}}) \cap \mathrm{L}(A_n)} \{R \subseteq \Sigma_{\mathrm{e}} : R \cup \Sigma_{\mathrm{i}} \in \mathrm{ref}(A_n, s)\} \\ = & \bigcup_{s \in \mathrm{p}_{\mathrm{e}}^{-1}(s_{\mathrm{e}}) \cap \mathrm{L}(A_n)} \{R \subseteq \Sigma_{\mathrm{e}} : (R \cup \Sigma_{\mathrm{i}}) \cap \Sigma_{\mathrm{e}} \in \mathrm{ref}(E, \mathrm{p}_{\mathrm{e}}(s))\} \\ = & \qquad [\mathrm{p}_{\mathrm{e}}(s) = s_{\mathrm{e}} \text{ and } (R \cup \Sigma_{\mathrm{i}}) \cap \Sigma_{\mathrm{e}} = R] \\ & \mathrm{ref}(E, s_{\mathrm{e}}). \end{aligned}$$

Hence for all $n \in \mathbb{N}, \mathrm{P_e}(A_n) \precsim E$ and as $E^\uparrow$ is the supremal of $\{A \in \Pi(\Sigma) : \mathrm{P_e}(A) \precsim E\}$ it follows that $A_n \precsim E^\uparrow$.

Second, it will be proven that $\mathrm{L}(E_{\mathrm{con}}) = \mathrm{L}(E^\uparrow)$. As all processes that reduce E have no trace not contained in $\mathrm{L}(E)$ it follows that $\mathrm{L}(E^\uparrow) \subseteq \mathrm{L}(E_{\mathrm{con}})$. As $\bigcup_{n\in\mathbb{N}} \Sigma_{\mathrm{i}}^n = \Sigma_{\mathrm{i}}^*$ and for all $n \in \mathbb{N}$ $\mathrm{L}(A_n) \subseteq \mathrm{L}(E^\uparrow)$, it follows that $\mathrm{L}(E_{\mathrm{con}}) = \bigcup_{n\in\mathbb{N}} \mathrm{L}(A_n) \subseteq \mathrm{L}(E^\uparrow)$. Hence $\mathrm{L}(E_{\mathrm{con}}) = \mathrm{L}(E^\uparrow)$.

Next it will be proven that for all $s \in \mathrm{L}(E_{\mathrm{con}}) = \mathrm{L}(E^\uparrow)$, $\mathrm{ref}(E_{\mathrm{con}}, s) \subseteq \mathrm{ref}(E^\uparrow, s)$. For all $s \in \mathrm{L}(E^\uparrow)$ there exists an $n \in \mathbb{N}$ such that $s \in A_n$. As $\mathrm{ref}(A_n, s) = \mathrm{ref}(E_{\mathrm{con}}, s)$ and $A_n \precsim E^\uparrow$, it follows that $\mathrm{ref}(E_{\mathrm{con}}, s) \subseteq \mathrm{ref}(E^\uparrow, s)$.

Finally, it will be proven that $\mathrm{ref}(E^\uparrow, s) \subseteq \mathrm{ref}(E_{\mathrm{con}}, s)$. Suppose the inclusion does not hold. Then there must exist a process B, and a refusal $R \in \mathrm{ref}(B, s)$ such that $\mathrm{P_e}(B) \precsim E$ and $R \notin \mathrm{ref}(E_{\mathrm{con}}, s)$. That is $\Sigma_{\mathrm{i}} \subseteq R$ and $R \cap \Sigma_{\mathrm{e}} \notin \mathrm{ref}(E, \mathrm{p_e}(s))$. But then it follows from the definition of projection that $\mathrm{P_e}(B) \not\precsim E$, which contradicts our assumption. Hence $\mathrm{ref}(E^\uparrow, s) \subseteq \mathrm{ref}(E_{\mathrm{con}}, s)$.

We have proven that $\mathrm{L}(E_{\mathrm{con}}) = \mathrm{L}(E^\uparrow)$ and for all $s \in \mathrm{L}(E^\uparrow)$, $\mathrm{ref}(E_{\mathrm{con}}, s) = \mathrm{ref}(E^\uparrow, s)$, so $E_{\mathrm{con}} = E^\uparrow$. □

Definition 4.13 Let G have a finite state space. Define $E_{\mathrm{br}}^\uparrow$ as the supremal of all legal implementations that are bounded recurrent with respect to G and E.

$$E_{\mathrm{br}}^\uparrow \quad = \quad \sup_{\precsim} \{A \in \Pi(\Sigma) : \mathrm{P_e}(A) \precsim E,\ A \text{ is bounded recurrent}\}.$$

Proposition 4.14 *Let G have a finite state space. Process $E_{\mathrm{br}}^\uparrow$ satisfies:*

$$\begin{aligned} \Sigma(E_{\mathrm{br}}^\uparrow) &= \Sigma, \\ \mathrm{L}(E_{\mathrm{br}}^\uparrow) &= \{s \in \Sigma^* : \mathrm{p_e}(s) \in \mathrm{L}(E) \text{ and } s \text{ is bounded recurrent}\}, \\ \mathrm{ref}(E_{\mathrm{br}}^\uparrow, s) &= \mathrm{ref}(E^\uparrow, s). \end{aligned}$$

This characterization can be used to construct $E_{\mathrm{br}}^\uparrow$.

Proof. Let E_{con} be the process defined by the expressions in the proposition. We have to prove that $E_{\mathrm{br}}^\uparrow = E_{\mathrm{con}}$. First it will be proven that $\mathrm{P_e}(E_{\mathrm{con}}) \precsim E$. The language part follows from $\mathrm{L}(\mathrm{P_e}(E_{\mathrm{con}})) = \mathrm{p_e}(\mathrm{L}(E_{\mathrm{con}})) = \mathrm{L}(E)$. For the refusal part let $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{L}(E_{\mathrm{con}}))$. If $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{div}(E_{\mathrm{con}}, \Sigma_{\mathrm{e}}))$ then, by Lemma 4.10, $\Sigma_{\mathrm{e}} \in \mathrm{ref}(E, s_{\mathrm{e}})$. So $\mathrm{ref}(\mathrm{P_e}(E_{\mathrm{con}}), s_{\mathrm{e}}) \subseteq \mathrm{ref}(E, s_{\mathrm{e}})$. If $s_{\mathrm{e}} \notin \mathrm{p_e}(\mathrm{div}(E_{\mathrm{con}}, \Sigma_{\mathrm{e}}))$ then,

$$\begin{aligned}
&\mathrm{ref}(\mathrm{P_e}(E_{\mathrm{con}}), s_{\mathrm{e}}) \\
=\;& \bigcup_{s\in \mathrm{p_e}^{-1}(s_{\mathrm{e}})\cap \mathrm{L}(E_{\mathrm{con}})} \mathrm{p_e}(\mathrm{ref}(E_{\mathrm{con}}, s)) \\
=\;& \bigcup_{s\in \mathrm{p_e}^{-1}(s_{\mathrm{e}})\cap \mathrm{L}(E_{\mathrm{con}})} \{R \subseteq \Sigma_{\mathrm{e}} : R \cup \Sigma_{\mathrm{i}} \in \mathrm{ref}(E_{\mathrm{con}}, s)\} \\
=\;& \bigcup_{s\in \mathrm{p_e}^{-1}(s_{\mathrm{e}})\cap \mathrm{L}(E_{\mathrm{con}})} \{R \subseteq \Sigma_{\mathrm{e}} : (R \cup \Sigma_{\mathrm{i}}) \cap \Sigma_{\mathrm{e}} \in \mathrm{ref}(E, \mathrm{p_e}(s))\} \\
=\;& \qquad\qquad [\mathrm{p_e}(s) = s_{\mathrm{e}} \text{ and } (R \cup \Sigma_{\mathrm{i}}) \cap \Sigma_{\mathrm{e}} = R] \\
&\mathrm{ref}(E, s_{\mathrm{e}}).
\end{aligned}$$

So, $\mathrm{P_e}(E_{\mathrm{con}}) \precsim E$. As E_{con} is bounded recurrent, it follows from Definition 4.13 that $E_{\mathrm{con}} \precsim E_{\mathrm{br}}^{\uparrow}$.

It remains to prove that $E_{\mathrm{br}}^{\uparrow} \precsim E_{\mathrm{con}}$. Suppose the relation does not hold. Then there must exist a process A such that $\mathrm{P_e}(A) \precsim E$, A is bounded recurrent, but A does not reduce E_{con}. As $\mathrm{p_e}(\mathrm{L}(A)) \subseteq \mathrm{L}(E)$ and A is bounded recurrent, it follows that $\forall s \in \mathrm{L}(A)$, $\mathrm{p_e}(s) \in \mathrm{L}(E)$ and s is bounded recurrent. So $\mathrm{L}(A) \subseteq \mathrm{L}(E_{\mathrm{con}})$. As $A \not\precsim E_{\mathrm{con}}$ there must exist an $s \in \mathrm{L}(A)$ and a $R \in \mathrm{ref}(A, s)$ such that $R \notin \mathrm{ref}(E_{\mathrm{con}}, s)$, i.e. $\Sigma_{\mathrm{i}} \subseteq R$ and $R \cap \Sigma_{\mathrm{e}} \notin \mathrm{ref}(E, \mathrm{p_e}(s))$. But then $\mathrm{P_e}(A) \not\precsim E$, which contradicts our assumptions. Hence $E_{\mathrm{br}}^{\uparrow} = E_{\mathrm{con}}$.
□

Theorem 4.15 *Let $G, S \in \Pi(\Sigma)$, $E \in \Pi(\Sigma_{\mathrm{e}})$, and $E_{\mathrm{br}}^{\uparrow}$ be constructed as above. Let G have a finite state space.*

$$G||S \precsim E_{\mathrm{br}}^{\uparrow} \iff \mathrm{P_e}(G||S) \precsim E \text{ and } G||S \text{ is bounded recurrent.}$$

Proof. ($G||S \precsim E_{\mathrm{br}}^{\uparrow} \Rightarrow G||S$ is bounded recurrent.) As $E_{\mathrm{br}}^{\uparrow}$ is bounded recurrent and $\mathrm{L}(G||S) \subseteq \mathrm{L}(E_{\mathrm{br}}^{\uparrow})$, it follows that $G||S$ is bounded recurrent.

($G||S \precsim E_{\mathrm{br}}^{\uparrow} \Rightarrow \mathrm{P_e}(G||S) \precsim E$.) In the proof of Proposition 4.14 we have shown that $E_{\mathrm{con}} = E_{\mathrm{br}}^{\uparrow}$ and that $\mathrm{P_e}(E_{\mathrm{con}}) \precsim E$. So $\mathrm{P_e}(E_{\mathrm{br}}^{\uparrow}) \precsim E$. And thus by Proposition 4.4 $\mathrm{P_e}(G||S) \precsim \mathrm{P_e}(E_{\mathrm{br}}^{\uparrow}) \precsim E$.

($\mathrm{P_e}(G||S) \precsim E$ and $G||S$ is bounded recurrent $\Rightarrow G||S \precsim E_{\mathrm{br}}^{\uparrow}$.) As $E_{\mathrm{br}}^{\uparrow}$ is the supremal element of the set $\{A \in \Pi(\Sigma) : \mathrm{P_e}(A) \precsim E$, A is bounded recurrent$\}$ it must hold that $G||S \precsim E_{\mathrm{br}}^{\uparrow}$. □

The following two results are proven in Section 4.7. Theorem 4.16 states that the control problem with partial specification can be converted to a basic control problem with full specification as defined and solved in Chapter 3.

Theorem 4.16 *Let $G \in \Pi(\Sigma)$ and $E \in \Pi(\Sigma_{\mathrm{e}})$. Let G have a finite state space. There exists a complete supervisor $S \in \Pi(\Sigma)$, such that $\mathrm{P_e}(G||S) \precsim E$ if and only if there exists a complete supervisor $S_{\mathrm{br}} \in \Pi(\Sigma)$ such that $G||S_{\mathrm{br}} \precsim E_{\mathrm{br}}^{\uparrow}$.*

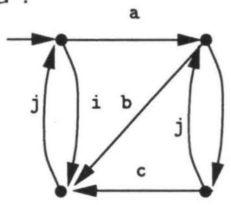
A condition for the existence of a complete supervisor S_{br} that solves $G||S_{\mathrm{br}} \precsim E^{\uparrow}_{\mathrm{br}}$ is given in Theorem 3.2. In [44] it was proven that the external language of the implementation is not restricted by the use of bounded recurrent implementations. Corollary 4.17 extends this results to the whole external behavior (the language and the refusal sets) of the implementation. Note that, $G||S_{\mathrm{br}} \precsim E^{\uparrow}_{\mathrm{br}}$ implies that $G||S_{\mathrm{br}}$ is bounded recurrent.

Corollary 4.17 *Let $G \in \Pi(\Sigma)$ and $E \in \Pi(\Sigma_{\mathrm{e}})$. Let G have a finite state space. Let $S \in \Pi(\Sigma)$ be a complete supervisor such that $\mathrm{P}_{\mathrm{e}}(G||S) \precsim E$. Then there exists a complete supervisor $S_{\mathrm{br}} \in \Pi(\Sigma)$ such that $\mathrm{P}_{\mathrm{e}}(G||S_{\mathrm{br}}) \precsim E$, $G||S_{\mathrm{br}}$ is bounded recurrent, and $\mathrm{P}_{\mathrm{e}}(G||S) = \mathrm{P}_{\mathrm{e}}(G||S_{\mathrm{br}})$.*

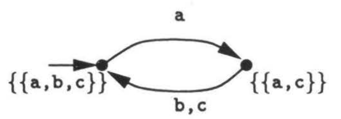
The next example will illustrate the followed approach.

Example 4.18 Let G and E be defined by the behavior state representations given below.

G :

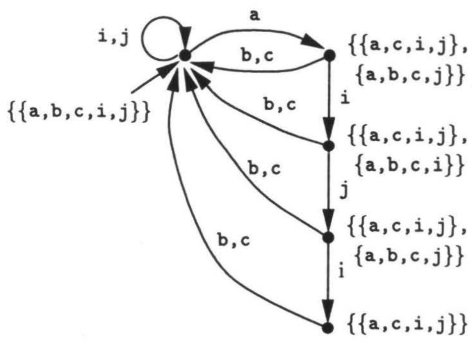


E :



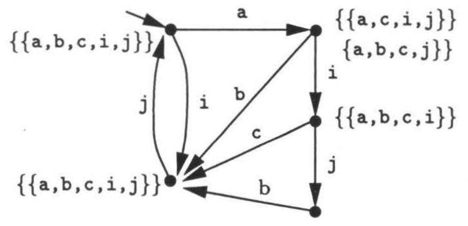
Let $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{i},\mathtt{j}\}$, $\Sigma_{\mathrm{e}} = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$, $\Sigma_{\mathrm{i}} = \{\mathtt{i},\mathtt{j}\}$, $\Sigma_{\mathrm{c}} = \Sigma$, and $\Sigma_{\mathrm{u}} = \emptyset$. As E can refuse the whole external event set in behavior state $[\varepsilon]_{\mathrm{E}}$, it is not necessary to bound the number of internal recurrences after traces that end in this state. After traces that end the other behavior state, E cannot refuse Σ_{e}, so the number of internal recurrences needs to be bounded. Below the behavior state representation of $E^{\uparrow}_{\mathrm{br}}$ is given.

$E^{\uparrow}_{\mathrm{br}}$:



The synthesis procedure results in the supremal supervisor $S^{\uparrow}$.

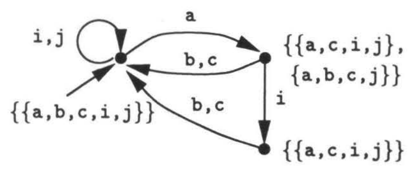
$S^{\uparrow}$:



Note that the trace **aic** is an element of $L(S^{\uparrow})$ as well as $L(G)$. So trace **ac** is an element of $L(P_e(G||S^{\uparrow}_{det}))$.

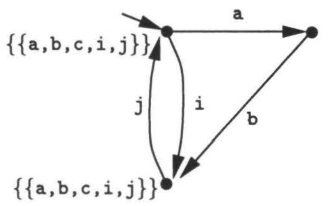
The system $E^{\uparrow}_{br(1)}$, given below, contains only traces that make at most one loop of internal events in G.

$E^{\uparrow}_{br(1)}$:



System $S^{\uparrow}_{br(1)}$ is the supremal supervisor corresponding to the specification $E^{\uparrow}_{br(1)}$.

$S^{\uparrow}_{br(1)}$:



Note that in this case **aic** $\notin L(S^{\uparrow}_{br(1)})$, so **ac** $\notin L(P_e(G||S^{\uparrow}_{br(1)}))$. One loop is not sufficient to guarantee that all external behavior is obtainable.

Complexity

If E and G are given by behavior state representations, then computing $E^{\uparrow}_{br}$ requires that each behavior state of E is replaced by either a behavior state with Σ_i self-loops if $\Sigma_e \in \text{ref}(E, s)$, or by a tree of behavior states, where on each path from root to leaf each behavior state of G occurs at most two times. The computation requires the set of states that are visited once and the set of states that are visited twice to be administrated. Worst case, all possible combinations of states can occur. This will result in complexity that is exponential in the size of the behavior state space of G and linear in the size of the behavior state space of E. Combining this with the supervisor synthesis algorithm from Chapter 3 will result in an algorithm which is exponential in

$|G|$ and linear in $|E|$. When unwinding a loop of internal events in G, the algorithm has to administrate only those states that are reachable by internal events. So the algorithm will be exponential in the sizes of the sets of states reachable by internal events. In most practical systems these sets will be much smaller than the whole state space. Therefore it is expected that the algorithm will behave better on practical systems than can be expected from the worst case analysis. Further research is needed to test the complexity of the algorithm in real life situations.

In [27] K. Inan presents the projected specification problem which is similar to the control problem with partial specification discussed in this chapter. Although Inan uses a nondeterministic supervisor as a finite representation of the, possibly infinite, set of solutions, the approach is based on languages. He does not consider the refusal properties of unbounded internal continuations (divergence).

Unlike methods based on languages, failure semantics has the ability to deal with phenomena, such as divergence, that occur when processes are partially observed. Correct handling of divergence requires that solutions are restricted to bounded recurrent supervisors. Note that the necessity of the bounded recurrence condition is not due to the use of failure semantics, but due to the nondeterministic properties of projected systems.

4.5. Externally Least Restrictive Solutions

Similar to the basic supervisory control problem, the supremal supervisor is not intended to be implemented directly. It is used as an intermediate result to describe the set of possible solutions. In Chapter 3 it was argued that a supervisor which is intended to be implemented should be least restrictive, deterministic, and its language should be contained in $\mathrm{L}(G)$. As a user can observe only the external behavior of the controlled system, only the external behavior needs to be least restrictive.

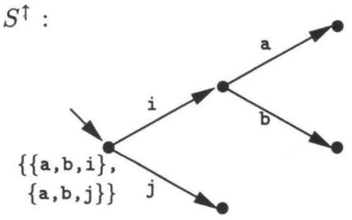
Definition 4.19 Supervisor $S^o \in \Pi(\Sigma)$, $S^o \underset{\sim}{\sqsubseteq} S^\uparrow$ is called *externally least restrictive* (w.r.t. G and $S^\uparrow$) if for all $S \in \Pi(\Sigma)$, $S \underset{\sim}{\sqsubseteq} S^\uparrow$ implies

$$\begin{array}{ll} \mathrm{L}(\mathrm{P_e}(G||S)) \subseteq \mathrm{L}(\mathrm{P_e}((G||S^o)) & \text{and} \\ \mathrm{ref}(\mathrm{P_e}(G||S^o), s) \subseteq \mathrm{ref}(\mathrm{P_e}(G||S), s) & \text{for all } s \in \mathrm{L}(\mathrm{P_e}(G||S)). \end{array}$$

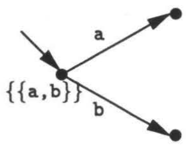
Unfortunately an externally least restrictive supervisor may be hard to find or even non-existent.

Example 4.20 Consider the supremal supervisor $S_1^\uparrow$ and the corresponding external behavior of the controlled system, where the uncontrolled system $G_1 = \mathrm{Det}(\mathrm{L}(S_1^\uparrow))$. (For simplicity reasons only the traces of $S_1^\uparrow$ contained in the language of G_1 are shown.)

$S^{\uparrow}$:

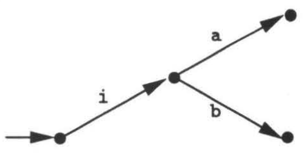


$\mathrm{P_e}(G_1||\mathrm{Det}(\mathrm{L}(S_1^{\uparrow})))$:

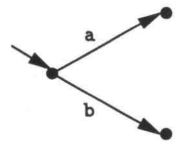


The deterministic least restrictive supervisor is, according to Proposition 3.7, equal to $\mathrm{Det}(\mathrm{L}(G_1||S_1^{\uparrow})) = \mathrm{Det}(\mathrm{L}(S_1^{\uparrow}))$. This supervisor is not externally least restrictive. Consider the supervisor S_1^o and the corresponding controlled system.

S_1^o :



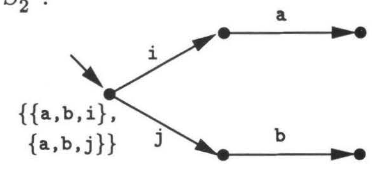
$\mathrm{P_e}(G_1||S_1^o)$:



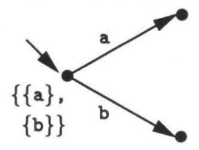
Comparing $\mathrm{P_e}(G_1||\mathrm{Det}(\mathrm{L}(S_1^{\uparrow})))$ and $\mathrm{P_e}(G_1||S_1^o)$ one can see that the external languages of the controlled systems are equal. As the second system refuses less than the first system, the latter is less restrictive than the first. For this control problem supervisor S_1^o is an externally least restrictive solution.

For some control problems there may not exist a externally least restrictive solution. Consider the supremal supervisor $S_2^{\uparrow}$ and the corresponding controlled system, where $G_2 = \mathrm{Det}(\mathrm{L}(S_2^{\uparrow}))$.

$S_2^{\uparrow}$:

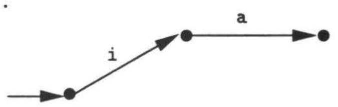


$\mathrm{P_e}(G_2||\mathrm{Det}(\mathrm{L}(S_2^{\uparrow})))$:

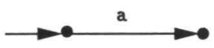


The deterministic least restrictive supervisor $\mathrm{Det}(\mathrm{L}(G_2||S_2^{\uparrow})) = \mathrm{Det}(\mathrm{L}(S_2^{\uparrow}))$ is not externally least restrictive. Consider the systems S_2 and $\mathrm{P_e}(G_2||S_2)$.

S_2^o :



$\mathrm{P_e}(G_2||S_2^o)$:



The language of $\mathrm{P_e}(G_2||\mathrm{Det}(\mathrm{L}(S_2^{\uparrow})))$ is strictly larger than the language of $\mathrm{P_e}(G_2||S_2)$. However, the refusal set $\mathrm{ref}(\mathrm{P_e}(G_2||S_2), \varepsilon)$ is strictly smaller than $\mathrm{ref}(\mathrm{P_e}(G_2||\mathrm{Det}(\mathrm{L}(S_2^{\uparrow}))), \varepsilon)$. So neither of the two systems is least restrictive. For this control problem no externally least restrictive solution exists.

If no externally least restrictive solution exists, then there are several, mutually uncomparable, externally minimal restrictive solutions. A supervisor is called externally minimal restrictive, it there does not exists a supervisor which is less restrictive.

Definition 4.21 Supervisor $S^o \in \Pi(\Sigma)$, $S^o \precsim S^\uparrow$ is called *externally minimal restrictive* if there does not exists an $S \in \Pi(\Sigma)$, $S \neq S^o$, $S \precsim S^\uparrow$ such that

$$\begin{array}{ll} \mathrm{L}(\mathrm{P_e}(G||S^o)) \subseteq \mathrm{L}(\mathrm{P_e}((G||S)) & \text{and} \\ \mathrm{ref}(\mathrm{P_e}(G||S), s) \subseteq \mathrm{ref}(\mathrm{P_e}(G||S^o), s) & \text{for all } s \in \mathrm{L}(\mathrm{P_e}(G||S^o)). \end{array}$$

Further research is needed to derive externally least restrictive or externally minimal restrictive solutions.

4.6. Task Completion Event

In Section 2.5 a framework which uses marking is compared to the framework based on failure semantics. Marking is used to indicate that a task is completed. If the system is in a marked state then it has completed a task. The system is called M-nonblocking if it can always reach a marked state. The completion of a task can also be indicated by an extra event. A task is completed if this event is executed.

Definition 4.22 Let the *task completion event* ($\mathtt{tc}$) be the special event to indicate that a task is completed. Process A is called nonblocking if

$$\{\mathtt{tc}\} \notin \mathrm{ref}(\mathrm{P_{tc}}(A), s), \quad \text{for all } s \in \Sigma_{\mathrm{tc}}^*,$$

where $\mathrm{P_{tc}}$ is the projection onto $\Sigma_{\mathrm{tc}} = \{\mathtt{tc}\}$.

Process A is nonblocking if $\mathrm{P_{tc}}(A)$ cannot refuse the task completion event. This means that always eventually the task completion event will be executed. This concept of nonblocking is somewhat different from the M-nonblocking concept. With M-nonblocking a system *can* always eventually complete a task. The difference is related to divergence. Consider the two systems shown in Figure 4.2. The encircled node indicates a marked state. System A_{m} is M-nonblocking, because from every state the marked state can be reached. The corresponding system with the task completion event is however blocking. Observe that an unbounded number of `coin` and `cookie` events can be executed. So when projected onto Σ_{tc}, the system can diverge. It can refuse to execute the task completion event.

If a system is M-nonblocking then it is not guaranteed that eventually a task will be completed. If it is nonblocking then it will always eventually complete a task.

Consider a specification E. Assume E is nonblocking. Let A be a process that implements E. So $A \precsim E$. Consider the following corollary, which is a direct consequence of Theorem 1 in [11].

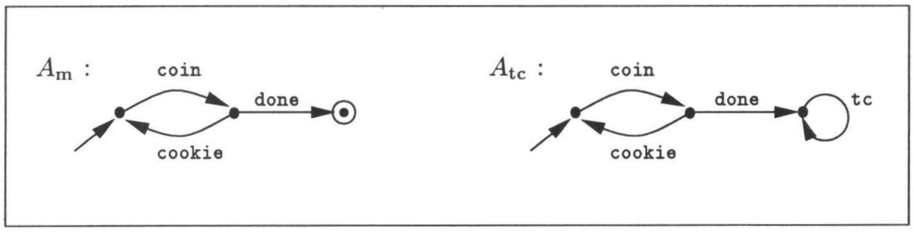


Figure 4.2: M-nonblocking supervisor and a corresponding blocking supervisor.

Corollary 4.23 *Let* $A, E \in \Pi(\Sigma)$.

$$A \sqsubseteq_{\sim} E \quad \Rightarrow \quad \mathrm{P}_{\mathrm{tc}}(A) \sqsubseteq_{\sim} \mathrm{P}_{\mathrm{tc}}(E).$$

As $\mathrm{P}_{\mathrm{tc}}(E)$ cannot refuse the task completion event, neither can $\mathrm{P}_{\mathrm{tc}}(A)$. Hence A will always eventually execute a task completion event. If the specification E is nonblocking then so is any implementation of E. An extra nonblocking requirement is not needed in the implementation relation. With marking such a requirement is needed. See Definition 2.20. This shows that the issue of blocking can be handled by a framework based on failure semantics.

Assume a specification with marked states is given. How can it be converted to a specification using the task completion event? Consider the following procedure on the behavior state representation of a process. The system is converted such that it executes a task completion event just before it enters a marked state. So, every time a marked state is reached also a task completion event is executed.

1. Add a new state for every marked state. These new states can refuse all event sets that do not contain the task completion event.

2. Redirect transitions entering a marked state to the corresponding new state.

3. Add transitions labeled with the task completion event from every new state to the corresponding marked state.

4. If the system is allowed to deadlock in a marked state, then add a self-loop, labeled with the task completion event, to this state.

The procedure is illustrated in Figure 4.3. Process E_{m} is the original system with marking. System E_{tc} is the corresponding system with the task completion event. As can be seen from the projection onto Σ_{tc} this system is nonblocking.

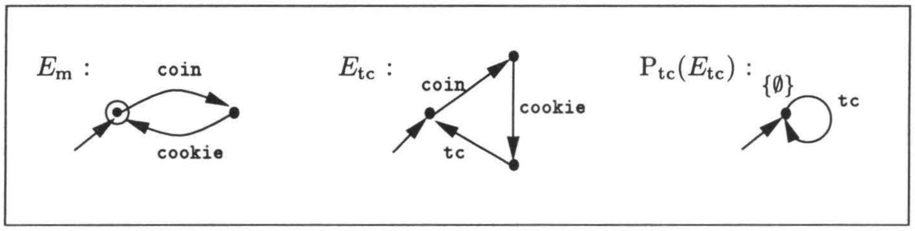


Figure 4.3: Non-blocking specifications.

4.7. Proof of Theorem 4.16 and Corollary 4.17

According to Theorem 4.15 a supervisor S_{br} solves $G||S \precsim E_{\mathrm{br}}^{\uparrow}$ if and only if $G||S_{\mathrm{br}}$ is bounded recurrent and $\mathrm{P_e}(G||S_{\mathrm{br}}) \precsim E$. So, in order to prove Theorem 4.16 it is necessary and sufficient to show that there exists a solution to the control problem with partial specification if and only if there exists a solution such that the controlled system is bounded recurrent. The if-part is trivial because a solution such that the controlled system is bounded recurrent, is a solution by itself. For the only-if part it will be shown that if there exists a supervisor S then we can construct a bounded recurrent supervisor S_{br}. Then also the controlled system $G||S_{\mathrm{br}}$ is bounded recurrent. In the rest of this section it will be assumed that G has a finite state space.

Definition 4.24 Let G, $S \in \Pi(\Sigma)$ and $E \in \Pi(\Sigma_{\mathrm{e}})$ be processes. Let S_G be the process S restricted to the language $\mathrm{L}(G)$.

$$S_G \quad = \quad S||\mathrm{Det}(\mathrm{L}(G)).$$

Lemma 4.25 *Let* $G \in \Pi(\Sigma)$. $G = \mathrm{Det}(\mathrm{L}(G))||G$.

Proof. The language part follows from

$$\mathrm{L}(\mathrm{Det}(\mathrm{L}(G))||G) = \mathrm{L}(G) \cap \mathrm{L}(G) = \mathrm{L}(G).$$

The refusal part follows from

$$\begin{aligned}
&R \in \mathrm{ref}(\mathrm{Det}(\mathrm{L}(G))||G, s)\\
&\iff \quad \exists R' \in \mathrm{ref}(\mathrm{Det}(\mathrm{L}(G)), s),\ R'' \in \mathrm{ref}(G, s),\ \text{s.t. } R = R' \cup R''\\
&\iff \quad \exists R' \subseteq \rho(\mathrm{L}(G), s),\ R'' \in \mathrm{ref}(G, s),\ \text{s.t. } R = R' \cup R''\\
&\iff \quad \exists R'' \in \mathrm{ref}(G, s) \text{ s.t. } R \subseteq \rho(\mathrm{L}(G), s) \cup R''\\
&\iff \qquad\qquad [G \text{ satisfies points } iv \text{ and } v \text{ of Definition 2.5}]\\
&\qquad\quad R \in \mathrm{ref}(G, s).
\end{aligned}$$

□

A direct consequence of Lemma 4.25 is that $G||S_G = G||\mathrm{Det}(\mathrm{L}(G))||S = G||S$.

The traces of a bounded recurrent process make at most two loops on the behavior state space of G. Consider a loop in G. There exist traces $s \in \mathrm{L}(G)$, $s_\mathrm{i} \in \Sigma_\mathrm{i}$ such that $[s]_G = [ss_\mathrm{i}]_G$. By Lemma 2.12 $ss_\mathrm{i} \in \mathrm{L}(G)$. If G can loop one time after trace s then it can also loop more times. For all $n \in \mathbb{N}$, $[s]_G = [ss_\mathrm{i}^n]_G$ and $ss_\mathrm{i}^n \in \mathrm{L}(G)$. So $s \in \mathrm{div}(G, \Sigma_\mathrm{e})$. Assume S is a supervisor that solves the supervisory control problem with partial observation. S is not necessarily bounded recurrent. Assume that $\Sigma_\mathrm{e} \notin \mathrm{ref}(E, \mathrm{p_e}(s))$. Then, as $\mathrm{P_e}(G||S) \sqsubseteq E$ it must hold, by the definition of projection, that $s \notin \mathrm{div}(G||S, \Sigma_\mathrm{e})$. As $s \in \mathrm{div}(G, \Sigma_\mathrm{e})$ it follows that $s \notin \mathrm{div}(S, \Sigma_\mathrm{e})$. There must exist an $n \in \mathbb{N}$ such that $ss_\mathrm{i}^n \in \mathrm{L}(S)$, but $ss_\mathrm{i}^{n+1} \notin \mathrm{L}(S)$. We will say that ss_i^n is in the last loop of S.

Definition 4.26 The Δ_i function gives all behavior states of process G that can be reached by internal events after trace s. All extensions have to be an element of $\mathrm{L}(S_G)$.

$$\begin{aligned}\Delta_\mathrm{i}(G, \mathrm{L}(S_G), s) &= \delta(G, s\Sigma_\mathrm{i}^+ \cap \mathrm{L}(S_G)) \\ &= \{[ss_\mathrm{i}]_G \in Q(G) : s_\mathrm{i} \in \Sigma_\mathrm{i}^+ \wedge ss_\mathrm{i} \in \mathrm{L}(S_G)\}.\end{aligned}$$

Trace $s \in \mathrm{L}(S_G)$ is in the last loop of S_G if $[s]_G \notin \Delta_\mathrm{i}(G, \mathrm{L}(S_G), s)$.

Definition 4.27 The set $\mathcal{Q}^{2\mathrm{nd}}(G, s)$ gives all behavior states of process G that have been visited at least twice by the last internal part of trace $s \in \mathrm{L}(G)$.

$$\begin{aligned}\mathcal{Q}^{2\mathrm{nd}}(G, s) = \{[s']_G \in Q^+(G) : \exists s', s'' \in \overline{s},\ s' \neq s'' \text{ s.t.} \\ \mathrm{p_e}(s'') = \mathrm{p_e}(s') = \mathrm{p_e}(s) \ \wedge\ [s']_G = [s'']_G\}\end{aligned}$$

Next, the process S_br will be constructed. As it will be bounded recurrent, it can make at most two loops of internal events. In the first loop it will copy all behavior of S_G after a trace that has the same projection and that reaches the same state in G. In the second loop (this is the last loop of S_br), process S_br will behave as S_G does in its last loop. The last loop of S_G is important, because it defines the traces after which all internal events can be refused. According to the definition of projection, these traces define the refusal sets of the projected implementation. Let $s_\mathrm{br} \in \mathrm{L}(S_\mathrm{br})$ be a trace such that $\mathcal{Q}^{2\mathrm{nd}}(G, s_\mathrm{br}) \neq \emptyset$. As S_br will be bounded recurrent, states in $\mathcal{Q}^{2\mathrm{nd}}(G, s_\mathrm{br})$ may not be visited a third time by internal continuations of s_br. Process S_br can only copy the behavior of S_G after a trace $s \in \mathrm{L}(S_G)$ if the internal continuations of s do not contain a state of $\mathcal{Q}^{2\mathrm{nd}}(G, s_\mathrm{br})$. That is $\Delta_\mathrm{i}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{2\mathrm{nd}}(G, s_\mathrm{br})$ has to be empty.

Definition 4.28 Define the set of traces in $\mathrm{L}(S_G)$ that *correspond* with a trace $s_\mathrm{br} \in \mathrm{L}(S_\mathrm{br})$ by

$$\begin{aligned}\mathrm{corr}(S_G, s_\mathrm{br}) = \{s \in \mathrm{L}(S_G) : [s]_G = [s_\mathrm{br}]_G,\ \mathrm{p_e}(s) = \mathrm{p_e}(s_\mathrm{br}) \text{ and} \\ s \notin \mathrm{div}(S_G, \Sigma_\mathrm{e}) \Rightarrow \Delta_\mathrm{i}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{2\mathrm{nd}}(G, s_\mathrm{br}) = \emptyset\}.\end{aligned}$$

S_{br} will be defined inductively. Let $\varepsilon \in \mathrm{L}(S_{\mathrm{br}})$. Let $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$. $s_{\mathrm{br}}\sigma \in \mathrm{L}(S_{\mathrm{br}})$ if

$$\exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } s\sigma \in \mathrm{L}(S_G),$$

and $R \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}})$ if

$$\exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R \in \mathrm{ref}(S_G, s).$$

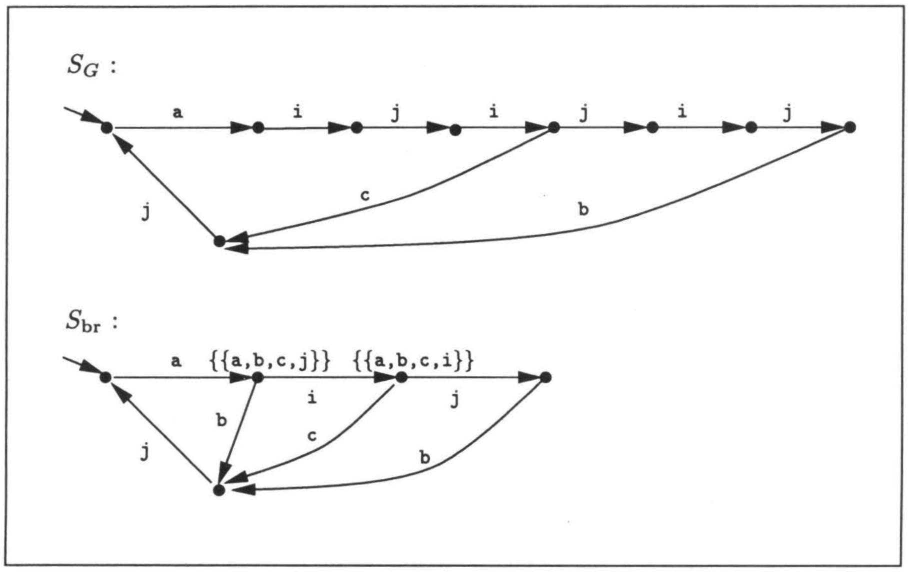


Figure 4.4: Construction of a bounded recurrent supervisor.

Example 4.29 Consider the uncontrolled system G and the specifcation E as defined in Example 4.18. Let S_G be the supervisor shown in Figure 4.4. From S_G the bounded recurrent supervisor S_{br}, also shown in Figure 4.4, can be constructed. The set of traces corresponding with trace $\mathtt{a} \in \mathrm{L}(S_{\mathrm{br}})$ is the set $\mathrm{corr}(S_G, \mathtt{a}) = \{\mathtt{a}, \mathtt{aij}, \mathtt{aijij}, \mathtt{aijijij}\}$. S_{br} copies the behavior of S_G after these traces. For example, because $\mathtt{aijijijb} \in \mathrm{L}(S_G)$, $\mathtt{ab} \in \mathrm{L}(S_{\mathrm{br}})$. And as $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{j}\} \in \mathrm{ref}(S_G, \mathtt{aij})$, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{j}\} \in \mathrm{ref}(S_{\mathrm{br}}, \mathtt{a})$. The set of corresponding traces of trace $\mathtt{aij} \in \mathrm{L}(S_{\mathrm{br}})$ contains only the trace $\mathtt{aijijij}$, because $[\mathtt{a}]_G \in \mathcal{Q}^{2\mathrm{nd}}(G, \mathtt{aij})$ and $\mathtt{aijijij}$ is the only trace for which $[\mathtt{a}]_G \notin \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s)$. So $\lambda(S_{\mathrm{br}}, \mathtt{aij}) = \lambda(S_G, \mathtt{aijijij})$ and $\mathrm{ref}(S_{\mathrm{br}}, \mathtt{aij}) = \mathrm{ref}(S_G, \mathtt{aijijij})$.

The following lemma states that S_{br} does not refuse more than S_G refuses after a corresponding trace.

Lemma 4.30 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. For all $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$, for all $R \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}})$, there exists an $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$ such that $R \in \mathrm{ref}(S_G, s)$.*

Proof. This result follows directly from the definition of $\mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}})$. □

Lemma 4.31 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. For all $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$, $\mathrm{corr}(S_G, s_{\mathrm{br}}) \neq \emptyset$.*

Proof. Let $s_{\mathrm{br}} = \varepsilon \in \mathrm{L}(S_{\mathrm{br}})$. As $\mathcal{Q}^{\mathrm{2nd}}(G, \varepsilon) = \emptyset$ it follows that $\varepsilon \in \mathrm{corr}(S_G, \varepsilon) \neq \emptyset$.

If $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ is not the empty trace, then s_{br} can be written as $v_{\mathrm{br}}\sigma$, with $v_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ and $\sigma \in \Sigma$. According to the definition of $\mathrm{L}(S_{\mathrm{br}})$, and because $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$, there must exist an $v \in \mathrm{corr}(S_G, v_{\mathrm{br}})$, such that $v\sigma \in \mathrm{L}(S_G)$. Let $s = v\sigma$. then it follows that for all $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ there exists an $s \in \mathrm{L}(S_G)$ such that $[s]_G = [v\sigma]_G = [v_{\mathrm{br}}\sigma]_G = [s_{\mathrm{br}}]_G$ and $\mathrm{p_e}(s) = \mathrm{p_e}(v\sigma) = \mathrm{p_e}(v_{\mathrm{br}}\sigma) = \mathrm{p_e}(s_{\mathrm{br}})$. We will prove by contradiction that for one of these traces $s \notin \mathrm{div}(S_G, \Sigma_{\mathrm{e}}) \Rightarrow \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$

Assume for all $s \in \mathrm{L}(S_G)$ with $[s]_G = [s_{\mathrm{br}}]_G$ and $\mathrm{p_e}(s) = \mathrm{p_e}(s_{\mathrm{br}})$, that (a) $s \notin \mathrm{div}(S_G, \Sigma_{\mathrm{e}})$ and (b) $\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \neq \emptyset$. Let $v_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ and $s_{\mathrm{i}} \in \Sigma_{\mathrm{i}}$ be such that $v_{\mathrm{br}}\sigma_{\mathrm{i}} = s_{\mathrm{br}}$. Note that $\sigma_{\mathrm{i}} \in \Sigma_{\mathrm{i}}$ because from $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \neq \emptyset$ it follows that the last internal part of s_{br} is not empty. From the definition of S_{br} it follows that there exists a $v \in \mathrm{corr}(S_G, v_{\mathrm{br}})$ such that $v\sigma_{\mathrm{i}} \in \mathrm{L}(S_G)$. Let $s = v\sigma_{\mathrm{i}}$. Then $[s]_G = [v\sigma_{\mathrm{i}}]_G = [v_{\mathrm{br}}\sigma_{\mathrm{i}}]_G = [s_{\mathrm{br}}]_G$ and $\mathrm{p_e}(s) = \mathrm{p_e}(v\sigma_{\mathrm{i}}) = \mathrm{p_e}(v_{\mathrm{br}}\sigma_{\mathrm{i}}) = \mathrm{p_e}(s_{\mathrm{br}})$, so by assumption (b)

$$\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \neq \emptyset. \tag{4.3}$$

As $v \in \overline{s}$ it follows from the definition of Δ_{i}

$$\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \subseteq \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v). \tag{4.4}$$

And from $s \notin \mathrm{div}(S_G, \Sigma_{\mathrm{e}})$, that $v \notin \mathrm{div}(S_G, \Sigma_{\mathrm{e}})$. Then, by $v \in \mathrm{corr}(S_G, v_{\mathrm{br}})$,

$$\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v) \cap \mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}}) = \emptyset. \tag{4.5}$$

By the definition of $\mathcal{Q}^{\mathrm{2nd}}$, and $s_{\mathrm{br}} = v_{\mathrm{br}}\sigma_{\mathrm{i}}$,

$$\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \subseteq \mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}}) \cup \{[s_{\mathrm{br}}]_G\}. \tag{4.6}$$

Combining (4.4) to (4.6) it folllows that

$$\begin{aligned}
&\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \\
&\quad\subseteq \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v) \cap (\mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}}) \cup \{[s_{\mathrm{br}}]_G\}) \\
&\quad= (\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v) \cap \mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}})) \cup (\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v) \cap \{[s_{\mathrm{br}}]_G\}) \\
&\quad= \emptyset \cup \{[s_{\mathrm{br}}]_G\} \\
&\quad= \{[s_{\mathrm{br}}]_G\}.
\end{aligned}$$

Then by (4.3)

$$\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \{[s_{\mathrm{br}}]_G\}. \tag{4.7}$$

It will be shown that an unbounded number of internal events can be executed.

$$\begin{array}{ll}
& \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \{[s_{\mathrm{br}}]_G\} \\
\Rightarrow & [s_{\mathrm{br}}]_G \in \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \\
\Rightarrow & \exists s_{\mathrm{i}} \in \Sigma_{\mathrm{i}}^{+} \text{ s.t. } s s_{\mathrm{i}} \in \mathrm{L}(S_G) \text{ and } [s s_{\mathrm{i}}]_G = [s_{\mathrm{br}}]_G \\
\Rightarrow & \hfill [\text{By assumption } (b))] \\
& \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s s_{\mathrm{i}}) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \neq \emptyset \\
\Rightarrow & \hfill [\text{By the same line of reasoning as in (4.3) to (4.7)}] \\
& \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s s_{\mathrm{i}}) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \{[s_{\mathrm{br}}]_G\} \\
\Rightarrow & [s_{\mathrm{br}}]_G \in \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s s_{\mathrm{i}}) \\
\Rightarrow & \exists s_{\mathrm{i}}' \in \Sigma_{\mathrm{i}}^{+} \text{ s.t. } s s_{\mathrm{i}} s_{\mathrm{i}}' \in \mathrm{L}(S_G) \text{ and } [s s_{\mathrm{i}} s_{\mathrm{i}}']_G = [s_{\mathrm{br}}]_G \\
\Rightarrow & \text{etc.}
\end{array}$$

It follows that an unbounded number of internal events can be executed. So $s \in \mathrm{div}(\mathrm{L}(S_G), \Sigma_{\mathrm{e}})$, which contradicts assumption (a). Hence there always exists an $s \in \mathrm{L}(S_G)$ such that either $s \notin \mathrm{div}(S_G, \Sigma_{\mathrm{e}})$ or $\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$. We can conclude that $\mathrm{corr}(S_G, s_{\mathrm{br}})$ is nonempty for all $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$. □

Lemma 4.32 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. Then $\mathrm{L}(S_{\mathrm{br}}) \subseteq \mathrm{L}(G)$.*

Proof. Let $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$. Then, by Lemma 4.31, there exists an $s \in \mathrm{L}(S_G)$ such that $[s]_G = [s_{\mathrm{br}}]_G$. As $\mathrm{L}(S_G) = \mathrm{L}(S) \cap \mathrm{L}(G) \subseteq \mathrm{L}(G)$ it follows that $s \in \mathrm{L}(G)$. Then, by Lemma 2.12 also $s_{\mathrm{br}} \in \mathrm{L}(G)$. □

Lemma 4.33 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. For all $s \in \mathrm{L}(S_G)$ there exists an $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ such that $\mathrm{p_e}(s) = \mathrm{p_e}(s_{\mathrm{br}})$, $[s]_G = [s_{\mathrm{br}}]_G$, and $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$.*

Proof. We will prove this lemma by induction on the length of the traces. The initial step is trivial because $\varepsilon \in \mathrm{L}(S_G)$, $\varepsilon \in \mathrm{L}(S_{\mathrm{br}})$, and $\mathcal{Q}^{\mathrm{2nd}}(G, \varepsilon) = \emptyset$.

For the inductive step assume $s \in \mathrm{L}(S_G)$ and $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ such that $\mathrm{p_e}(s) = \mathrm{p_e}(s_{\mathrm{br}})$, $[s]_G = [s_{\mathrm{br}}]_G$, and $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$. Let $s\sigma \in \mathrm{L}(S_G)$. We have to prove that there exists a $v_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ such that $\mathrm{p_e}(s\sigma) = \mathrm{p_e}(v_{\mathrm{br}})$, $[s\sigma]_G = [v_{\mathrm{br}}]_G$, and $\mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}}) = \emptyset$. As $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$, it follows that $\Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), s) \cap \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$. Then, by the definition of $\mathrm{L}(S_{\mathrm{br}})$, $s\sigma \in \mathrm{L}(S_G)$ implies $s_{\mathrm{br}}\sigma \in \mathrm{L}(S_{\mathrm{br}})$. From the definition of $\mathcal{Q}^{\mathrm{2nd}}$ it follows that

$\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}\sigma) \subseteq \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) \cup \{[s_{\mathrm{br}}\sigma]_G\}$. It was assumed that $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$, so $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}\sigma)$ contains at most one element.

If $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}\sigma) = \emptyset$ then $v_{\mathrm{br}} = s_{\mathrm{br}}\sigma$ satisfies the necessary conditions for the inductive step,

If $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}\sigma) = \{[s_{\mathrm{br}}\sigma]_G\}$ then the behavior state $[s_{\mathrm{br}}\sigma]_G$ has been visited at least twice by the last internal part of $s_{\mathrm{br}}\sigma$. Hence $[s_{\mathrm{br}}\sigma]_G$ has been visited at least once by the last internal part of s_{br}. So, there exists a $v_{\mathrm{br}} \in \overline{s_{\mathrm{br}}}$ (Note: $\overline{s_{\mathrm{br}}}$, *not* $\overline{s_{\mathrm{br}}\sigma}$), such that $[v_{\mathrm{br}}]_G = [s_{\mathrm{br}}\sigma]_G$ and $\mathrm{p_e}(v_{\mathrm{br}}) = \mathrm{p_e}(s_{\mathrm{br}}\sigma)$. It was assumed that $[s_{\mathrm{br}}]_G = [s]_G$, so $[v_{\mathrm{br}}]_G = [s_{\mathrm{br}}\sigma]_G = [s\sigma]_G$. As $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}\sigma)$ is not empty it follows that $\sigma \in \Sigma_{\mathrm{i}}$. So $\mathrm{p_e}(v_{\mathrm{br}}) = \mathrm{p_e}(s_{\mathrm{br}}\sigma) = \mathrm{p_e}(s\sigma)$. As v_{br} is a prefix of s_{br}, and $\mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$, it follows from the definition of $\mathcal{Q}^{\mathrm{2nd}}$ that $\mathcal{Q}^{\mathrm{2nd}}(G, v_{\mathrm{br}}) \subseteq \mathcal{Q}^{\mathrm{2nd}}(G, s_{\mathrm{br}}) = \emptyset$. Hence v_{br} satisfies the necessary conditions for the inductive step. □

Lemma 4.34 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. Then S_{br} is a process.*

Proof. It follows from the definition of synchronous composition that S_G is a process. We have to prove that S_{br} satisfies the five conditions in Definition 2.5. Points *i* and *ii* follow directly from the construction of S_{br}.

From Lemma 4.31 it follows that for all $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$, $\mathrm{corr}(S_G, s_{\mathrm{br}}) \neq \emptyset$. Let $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$. As S_G is a process, $\emptyset \in \mathrm{ref}(S_G, s)$. Then, by construction of $\mathrm{ref}(S_{\mathrm{br}})$ also $\emptyset \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}})$. So S_{br} satisfies point *iii*.

Let $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$. Then

$$\begin{aligned} & R \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}}) \wedge R' \subseteq R \\ & \quad \Rightarrow \quad \exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R \in \mathrm{ref}(S_G, s) \wedge R' \subseteq R \\ & \quad \Rightarrow \quad \exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R' \in \mathrm{ref}(S_G, s) \\ & \quad \Rightarrow \quad R' \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}}). \end{aligned}$$

So S_{br} satisfies point *iv*. From the construction of $\mathrm{L}(S_{\mathrm{br}})$ it follows that for all $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$

$$\lambda(\mathrm{L}(S_G), s) \subseteq \lambda(\mathrm{L}(S_{\mathrm{br}}), s_{\mathrm{br}}) \quad \Rightarrow \quad \rho(\mathrm{L}(S_{\mathrm{br}}), s_{\mathrm{br}}) \subseteq \rho(\mathrm{L}(S), s).$$

Then

$$\begin{aligned} & R \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}}) \\ & \quad \Rightarrow \quad \exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R \in \mathrm{ref}(S_G, s) \\ & \quad \Rightarrow \quad \exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R \cup \rho(\mathrm{L}(S_G), s) \in \mathrm{ref}(S_G, s) \\ & \quad \Rightarrow \quad \exists s \in \mathrm{corr}(S_G, s_{\mathrm{br}}) \text{ s.t. } R \cup \rho(\mathrm{L}(S_{\mathrm{br}}), s_{\mathrm{br}}) \in \mathrm{ref}(S_G, s) \\ & \quad \Rightarrow \quad R \cup \rho(\mathrm{L}(S_{\mathrm{br}}), s_{\mathrm{br}}) \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}}) \end{aligned}$$

So S_{br} satisfies point *v*. □

Lemma 4.35 *Let $S \in \Pi(\Sigma)$ and S be complete. Let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. Then S_{br} is complete.*

Proof. Let $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ and let $R \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}})$. By Lemma 4.30 there exists an $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$ such that $R \in \mathrm{ref}(S_G, s)$.

$$\begin{aligned} & R \in \mathrm{ref}(S_G, s) \\ & \quad \Rightarrow \quad \exists R_{\mathrm{s}} \in \mathrm{ref}(S_G, s),\ R_{\mathrm{g}} \in \mathrm{ref}(\mathrm{Det}(\mathrm{L}(G)), s) \text{ s.t. } R = R_{\mathrm{s}} \cup R_{\mathrm{g}} \\ & \quad \Rightarrow \quad \exists R_{\mathrm{s}} \in \mathrm{ref}(S_G, s) \text{ s.t. } R \subseteq R_{\mathrm{s}} \cup \rho(\mathrm{L}(G), s) \end{aligned}$$

As S_G is complete it follows that

$$\begin{aligned} R \cap \Sigma_{\mathrm{uc}} & \subseteq (R_{\mathrm{s}} \cup \rho(\mathrm{L}(G), s)) \cap \Sigma_{\mathrm{uc}} \\ & = (R_{\mathrm{s}} \cap \Sigma_{\mathrm{uc}}) \cup (\rho(\mathrm{L}(G), s) \cap \Sigma_{\mathrm{uc}}) \\ & \subseteq \rho(\mathrm{L}(G), s) \end{aligned}$$

As $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$ it follows that $[s]_G = [s_{\mathrm{br}}]_G]$. So $R \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G), s_{\mathrm{br}})$. Hence S_{br} is complete. □

Lemma 4.36 *Let $S \in \Pi(\Sigma)$ such that $\mathrm{P}_{\mathrm{e}}(G||S) \precsim E$. Let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. Then S_{br} is bounded recurrent.*

Proof. Assume S_{br} is not bounded recurrent. Then there exists an $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ such that $\Sigma_{\mathrm{e}} \notin \mathrm{ref}(E, \mathrm{p}_{\mathrm{e}}(s_{\mathrm{br}}))$ and $\mathrm{r}_{\mathrm{i}}(s_{\mathrm{br}}) \geq 3$. It follows that s_{br} can be written as $v_{\mathrm{br}}\sigma_{\mathrm{i}}$, where $v_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ and $\sigma_{\mathrm{i}} \in \Sigma_{\mathrm{i}}$. The behavior state $[s_{\mathrm{br}}]_G$ is visited at least three times by the last internal part of s_{br}. One visit is by the trace s_{br} itself, so that the behavior state $[s_{\mathrm{br}}]_G$ is visited at least twice by the last internal part of trace v_{br}. It follows that $[s_{\mathrm{br}}]_G \in \mathcal{Q}^{2\mathrm{nd}}(G, v_{\mathrm{br}})$. According to Lemma 4.31 there exists an $v \in \mathrm{L}(S_G)$ such that $[v]_G = [v_{\mathrm{br}}]_G$ and $\mathrm{p}_{\mathrm{e}}(v) = \mathrm{p}_{\mathrm{e}}(v_{\mathrm{br}})$. If $v\sigma_{\mathrm{i}} \in \mathrm{L}(S_G)$ then $[s_{\mathrm{br}}]_G = [v\sigma_{\mathrm{i}}]_G \in \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v)$. Hence for all $v \in \mathrm{L}(S_G)$ such that $v\sigma_{\mathrm{i}} \in \mathrm{L}(S_G)$, $[v]_G = [v_{\mathrm{br}}]_G$, and $\mathrm{p}_{\mathrm{e}}(v) = \mathrm{p}_{\mathrm{e}}(v_{\mathrm{br}})$, we have that $[s_{\mathrm{br}}]_G \in \Delta_{\mathrm{i}}(G, \mathrm{L}(S_G), v) \cap \mathcal{Q}^{2\mathrm{nd}}(G, v_{\mathrm{br}}) \neq \emptyset$.

But $s_{\mathrm{br}} = v_{\mathrm{br}}\sigma_{\mathrm{i}} \in \mathrm{L}(S_{\mathrm{br}})$. According to the definition of $\mathrm{L}(S_{\mathrm{br}})$ and $\mathrm{corr}(S_G, s_{\mathrm{br}})$, it must hold that there exists an $v \in \mathrm{L}(S_G)$ such that $v\sigma_{\mathrm{i}} \in \mathrm{L}(S_G)$, $[v]_G = [v_{\mathrm{br}}]_G$, $\mathrm{p}_{\mathrm{e}}(v) = \mathrm{p}_{\mathrm{e}}(v_{\mathrm{br}})$, and $v \in \mathrm{div}(S_G, \Sigma_{\mathrm{e}})$. Then, as $\mathrm{L}(S_G) \subseteq \mathrm{L}(G)$ also $v \in \mathrm{div}(G||S_G, \Sigma_{\mathrm{e}})$. So $\Sigma_{\mathrm{e}} \in \mathrm{ref}(\mathrm{P}_{\mathrm{e}}(G||S_G), \mathrm{p}_{\mathrm{e}}(v))$. As $\mathrm{P}_{\mathrm{e}}(G||S_G) = \mathrm{P}_{\mathrm{e}}(G||S) \precsim E$ it follows that $\Sigma_{\mathrm{e}} \in \mathrm{ref}(E, \mathrm{p}_{\mathrm{e}}(v)$. But this contradicts our assumptions that $\Sigma_{\mathrm{e}} \notin \mathrm{ref}(E, \mathrm{p}_{\mathrm{e}}(s_{\mathrm{br}}))$ and $\mathrm{p}_{\mathrm{e}}(v) = \mathrm{p}_{\mathrm{e}}(v_{\mathrm{br}}) = \mathrm{p}_{\mathrm{e}}(v_{\mathrm{br}}\sigma_{\mathrm{i}}) = \mathrm{p}_{\mathrm{e}}(s_{\mathrm{br}})$.

We can conclude that S_{br} is bounded recurrent. □

Lemma 4.37 *Let $S \in \Pi(\Sigma)$ and let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. Then*

$$\mathrm{P_e}(G||S_{\mathrm{br}}) \quad = \quad \mathrm{P_e}(G||S_G).$$

Proof. ($\mathrm{P_e}(G||S_{\mathrm{br}}) \precsim \mathrm{P_e}(G||S_G)$.) First we will prove that $\mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}})) \subseteq \mathrm{L}(\mathrm{P_e}(G||S_G))$.

Let $s_{\mathrm{e}} \in \mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}}))$. Let $s_{\mathrm{br}} \in \mathrm{L}(G) \cap \mathrm{L}(S_{\mathrm{br}})$ such that $p_{\mathrm{e}}(s_{\mathrm{br}}) = s_{\mathrm{e}}$. By Lemma 4.31 $corr(S_G, s_{\mathrm{br}})$ is not empty. Let $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$. Then by the definition of $\mathrm{corr}(S_G, s_{\mathrm{br}}\ \mathrm{p_e}(s) = \mathrm{p_e}(s_{\mathrm{br}}) = s_{\mathrm{e}})$. As $\mathrm{L}(S_G) \subseteq \mathrm{L}(G)$ it follows that $s \in \mathrm{L}(G||S_G)$. Hence $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{L}(G||S_G)) = \mathrm{L}(\mathrm{P_e}(G||S_G))$.
It remains to prove that for all $s \in \mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}}))$, $\mathrm{ref}(\mathrm{P_e}(G||S_{\mathrm{br}}), s_{\mathrm{e}}) \subseteq \mathrm{ref}(\mathrm{P_e}(G||S_G, s_{\mathrm{e}})$. Let $s_{\mathrm{e}} \in \mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}}))$. We will distinguish two cases: $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{div}(G||S_{\mathrm{br}}, \Sigma_{\mathrm{e}}))$ and $s_{\mathrm{e}} \notin \mathrm{p_e}(\mathrm{div}(G||S_{\mathrm{br}}, \Sigma_{\mathrm{e}}))$. Let $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{div}(G||S_{\mathrm{br}}, \Sigma_{\mathrm{e}}))$. By Lemma 4.36 S_{br} is bounded recurrent. As $\mathrm{L}(G||S_{\mathrm{br}}) \subseteq \mathrm{L}(S_{\mathrm{br}})$, $G||S_{\mathrm{br}}$ is also bounded recurent. Then from Lemma 4.10 it follows that $\Sigma_{\mathrm{e}} \in \mathrm{ref}(E, s_{\mathrm{e}})$. So $\mathrm{ref}(G||S_{\mathrm{br}}, s_{\mathrm{e}}) \subseteq \mathrm{ref}(E, s_{\mathrm{e}})$.

If $s_{\mathrm{e}} \notin \mathrm{p_e}(\mathrm{div}(G||S_{\mathrm{br}}, \Sigma_{\mathrm{e}}))$, then

$$\begin{aligned}
& R \in \mathrm{ref}(\mathrm{P_e}(G||S_{\mathrm{br}}), s_{\mathrm{e}}) \\
\Rightarrow\ & \exists s_{\mathrm{br}} \in \mathrm{L}(G||S_{\mathrm{br}}) \cap \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \text{ s.t. } R \in \mathrm{p_e}(\mathrm{ref}(G||S_{\mathrm{br}}, s_{\mathrm{br}})) \\
\Rightarrow\ & \exists s_{\mathrm{br}} \in \mathrm{L}(G||S_{\mathrm{br}}) \cap \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \text{ s.t. } R \cup \Sigma_{\mathrm{i}} \in \mathrm{ref}(G||S_{\mathrm{br}}, s_{\mathrm{br}}) \\
\Rightarrow\ & \exists s_{\mathrm{br}} \in \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \cap \mathrm{L}(G||S_{\mathrm{br}}),\ R_{\mathrm{g}} \in \mathrm{ref}(G, s),\ R_{\mathrm{s}} \in \mathrm{ref}(S_{\mathrm{br}}, s_{\mathrm{br}}) \\
& \text{s.t. } R_{\mathrm{g}} \cap R_{\mathrm{s}} = R \cup \Sigma_{\mathrm{i}}.
\end{aligned}$$

By Lemma 4.30 there exists a $s \in \mathrm{corr}(S_G, s_{\mathrm{br}})$ such that $R_{\mathrm{s}} \in \mathrm{ref}(S_G, s)$. It follows from $[s]_G = [s_{\mathrm{br}}]_G$ that $\mathrm{ref}(G, s) = \mathrm{ref}(G, s_{\mathrm{br}})$. So

$$\begin{aligned}
\Rightarrow\ & \exists s \in \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \cap \mathrm{L}(G||S_G),\ R_{\mathrm{g}} \in \mathrm{ref}(G, s),\ R_{\mathrm{s}} \in \mathrm{ref}(S_G, s) \\
& \text{s.t. } R_{\mathrm{g}} \cap R_{\mathrm{s}} = R \cup \Sigma_{\mathrm{i}} \\
\Rightarrow\ & \exists s \in \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \cap \mathrm{L}(G||S_G) \text{ s.t. } R \cup \Sigma_{\mathrm{i}} \in \mathrm{ref}(G||S_G, s) \\
\Rightarrow\ & \exists s \in \mathrm{p_e^{-1}}(s_{\mathrm{e}}) \cap \mathrm{L}(G||S_G) \text{ s.t. } R \in \mathrm{p_e}(\mathrm{ref}(G||S_G, s)) \\
\Rightarrow\ & R \in \mathrm{ref}(\mathrm{P_e}(G||S_G), s_{\mathrm{e}})
\end{aligned}$$

Hence $\mathrm{ref}(\mathrm{P_e}(G||S_{\mathrm{br}}), s_{\mathrm{e}}) \subseteq \mathrm{ref}(\mathrm{P_e}(G||S_G), s_{\mathrm{e}})$ and it can be concluded that $\mathrm{P_e}(G||S_{\mathrm{br}}) \precsim \mathrm{P_e}(G||S_G)$.

($\mathrm{P_e}(G||S_G) \precsim \mathrm{P_e}(G||S_{\mathrm{br}})$.) First it will be proven that $\mathrm{L}(\mathrm{P_e}(G||S_G)) \subseteq \mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}}))$.

Let $s_{\mathrm{e}} \in \mathrm{L}(\mathrm{P_e}(G||S_G))$. Let $s \in \mathrm{L}(G) \cap \mathrm{L}(S_G)$ such that $p_{\mathrm{e}}(s) = s_{\mathrm{e}}$. By Lemma 4.33, there exists a trace $s_{\mathrm{br}} \in \mathrm{L}(S_{\mathrm{br}})$ such that $\mathrm{p_e}(s_{\mathrm{br}}) = \mathrm{p_e}(s) = s_{\mathrm{e}})$. According to Lemma 4.32 $\mathrm{L}(S_{\mathrm{br}}) \subseteq \mathrm{L}(G)$. So $s_{\mathrm{br}} \in \mathrm{L}(G||S_{\mathrm{br}})$. And thus $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{L}(G||S_{\mathrm{br}})) = \mathrm{L}(\mathrm{P_e}(G||S_{\mathrm{br}}))$.

It remains to prove that for for all $s \in \mathrm{L}(\mathrm{P_e}(G||S_G))$, $\mathrm{ref}(\mathrm{P_e}(G||S_G), s_{\mathrm{e}}) \subseteq \mathrm{ref}(\mathrm{P_e}(G||S_{\mathrm{br}}, s_{\mathrm{e}})$. Let $s_{\mathrm{e}} \in \mathrm{L}(\mathrm{P_e}(G||S_G))$. We will consider two cases: $s_{\mathrm{e}} \in \mathrm{p_e}(\mathrm{div}(G||S_G, \Sigma_{\mathrm{e}}))$ and $s_{\mathrm{e}} \notin \mathrm{p_e}(\mathrm{div}(G||S_G, \Sigma_{\mathrm{e}}))$.

Let $s_e \in p_e(\text{div}(G||S_G, \Sigma_e))$. Then there exists a trace $s \in L(G||S_G)$ such that $p_e(s) = s_e$ and $s \in \text{div}(G||S_G, \Sigma_e)$. As $L(S_G) \subseteq L(G)$, $s \in \text{div}(S_G, \Sigma_e)$. By Lemma 4.33 there exists a trace $s_{br} \in L(S_{br})$ such that $[s]_G = [s_{br}]_G$ and $p_e(s_{br}) = p_e(s) = s_e$. It follows from the definition of $L(S_{br})$ that for all $ss_i \in L(S_G)$ with $s_i \in \Sigma_i^*$, $s_{br}s_i \in L(S_{br})$. So $s_{br} \in \text{div}(S_{br}, \Sigma_e)$. By Lemma 4.32 $L(S_{br}) \subseteq L(G)$, so $s_{br} \in \text{div}(G||S_{br}, \Sigma_e)$. And thus $s_e \in p_e(\text{div}(G||S_{br}, \Sigma_e))$. If $s_e \in p_e(\text{div}(G||S_G, \Sigma_e))$ then

$$\text{ref}(P_e(G||S_G), s_e) = 2^{\Sigma_e} = \text{ref}(P_e(G||S_{br}), s_e).$$

If $s_e \notin p_e(\text{div}(G||S_G, \Sigma_e))$, then

$$\begin{aligned}
& R \in \text{ref}(P_e(G||S_G), s_e) \\
\Rightarrow\quad & \exists s \in p_e^{-1}(s_e) \cap L(G||S_G) \text{ s.t. } R \in p_e(\text{ref}(G||S_G, s)) \\
\Rightarrow\quad & \exists s \in p_e^{-1}(s_e) \cap L(G||S_G) \text{ s.t. } R \cup \Sigma_i \in \text{ref}(G||S_G, s) \\
\Rightarrow\quad & \exists s \in p_e^{-1}(s_e) \cap L(G||S_G),\ R_g \in \text{ref}(G, s),\ R_s \in \text{ref}(S_G, s) \\
& \text{s.t. } R_g \cap R_s = R \cup \Sigma_i.
\end{aligned}$$

By Lemma 4.33 there exists a $s_{br} \in p_e^{-1}(s_e) \cap L(S_{br}) \subseteq L(G)$ such that $[s]_G = [s_{br}]_G$ and $\mathcal{Q}^{2nd}(G, s_{br}) = \emptyset$. Then it follows from the definition of S_{br} that $\text{ref}(S_G, s) \subseteq \text{ref}(S_{br}, s_{br})$. From $[s]_G = [s_{br}]_G$ it follows that $\text{ref}(G, s) = \text{ref}(G, s_{br})$. So

$$\begin{aligned}
& \exists s_{br} \in p_e^{-1}(s_e) \cap L(G||S_{br}),\ R_g \in \text{ref}(G, s_{br}),\ R_s \in \text{ref}(S_{br}, s_{br}) \\
& \text{s.t. } R_g \cap R_s = R \cup \Sigma_i \\
\Rightarrow\quad & \exists s_{br} \in p_e^{-1}(s_e) \cap L(G||S_{br}) \text{ s.t. } R \cup \Sigma_i \in \text{ref}(G||S_{br}, s_{br}) \\
\Rightarrow\quad & \exists s_{br} \in p_e^{-1}(s_e) \cap L(G||S_{br}) \text{ s.t. } R \in p_e(\text{ref}(G||S_{br}, s_{br})) \\
\Rightarrow\quad & R \in \text{ref}(P_e(G||S_{br}), s_e)
\end{aligned}$$

Hence $\text{ref}(P_e(G||S_G), s_e) \subseteq \text{ref}(P_e(G||S_{br}), s_e)$ and it can be concluded that $P_e(G||S_G) \sqsubseteq\!\!\!\!\!\!\sim\; P_e(G||S_{br})$. □

Proof. (*Theorem 4.16 and Corollary 4.17*)

(only-if part and Corollary 4.17) Let $S \in \Pi(\Sigma)$ be a complete supervisor such that $P_e(G||S) \precsim E$. Let S_G and S_{br} be constructed according to Definitions 4.24 and 4.28. From Lemmas 4.34, 4.35, and 4.36 it follows that S_{br} is a complete and bounded recurrent supervisor. Then $G||S_{br}$ is bounded recurrent, because $L(G||S_{br}) \subseteq L(S_{br})$. From Lemma 4.37 it follows that

$$P_e(G||S_{br}) = P_e(G||S_G) = P_e(G||S) \precsim E.$$

Then by Theorem 4.15 $G||S_{br} \precsim E_{br}^{\uparrow}$.

(if-part) This follows directly from Theorem 4.15. □

Chapter 5
Partial Observations

There are cases in which a supervisor cannot observe all events in the system. For instance an error can occur inside the system which cannot be observed. But the effects may be relevant for the behavior of the system, so we want to include the error event in the system. As the error cannot be observed by the supervisor, it is excluded from the event set of the supervisor. In the sequel let $\Sigma_o \subseteq \Sigma$ denote the set of events that are observed by the supervisor and let $\Sigma_{uo} = \Sigma - \Sigma_o$ be the set of unobservable events.

The supervisory control problem with partial observation for deterministic system has been treated by F. Lin and M. W. Wonham [36], and by R. Cieslak, C. Desclaux, A. S. Fawaz and P. Varaiya [13]. They showed that the control problem is much harder if the set of controllable events is not contained in the set of observable events. In Section 5.2 it will be shown that all discrete event control problems can be modeled such that all controllable events are observable. This result is related to the nondeterministic supervisor proposed K. Inan [27]. Yet, our approach illustrates more clearly the fundamental concepts underlying the modeling choices for discrete event control problems. In Section 5.1 it will be shown that the supervisory control problem with partial observation has a straightforward solution if all controllable events are observable.

5.1. Supervisory Control Problem with Partial Observation

In Section 2.2 the synchronous composition of two processes with the same alphabet is defined. For partial observation it is necessary to define the synchronous composition of two processes with different alphabets.

Definition 5.1 The *synchronous composition* of processes A and B is the process $A||B$ defined by the following equations, where $\mathrm{p_a}$ and $\mathrm{p_b}$ denote the projection on event sets $\Sigma(A)$ and $\Sigma(B)$ respectively.

$$\begin{aligned}
\Sigma(A||B) &= \Sigma(A) \cup \Sigma(B), \\
\mathrm{L}(A||B) &= \{s \in \Sigma(A||B)^* : \mathrm{p_a}(s) \in \mathrm{L}(A) \text{ and } \mathrm{p_b}(s) \in \mathrm{L}(B)\} \\
&= \mathrm{p_a^{-\uparrow}}(\mathrm{L}(A)) \cap \mathrm{p_b^{-\uparrow}}(\mathrm{L}(B)) \cap \Sigma(A||B)^*, \\
\mathrm{ref}(A||B, s) &= \{R_\mathrm{a} \cup R_\mathrm{b} : R_\mathrm{a} \in \mathrm{ref}(A, \mathrm{p_a}(s)),\ R_\mathrm{b} \in \mathrm{ref}(B, \mathrm{p_b}(s))\}.
\end{aligned}$$

Note that if $\Sigma(A) = \Sigma(B)$ then the definition is equivalent to Definition 2.9

Proposition 2.16 still holds with this new definition of synchronous composition. All the nice properties of the framework, concerning deadlock behavior and modular control, remain valid.

Definition 5.2 Let the uncontrolled system $G \in \Pi(\Sigma)$, the specification $E \in \Pi(\Sigma)$, and the set of observable events $\Sigma_\mathrm{o} \subseteq \Sigma$ be given. The *supervisory control problem with partial observation* is to find a supervisor $S_\mathrm{o} \in \Pi(\Sigma_o)$, such that $G||S_\mathrm{o} \precsim E$.

Event σ is called *enabled* by supervisor S if σ can be executed in the controlled system $G||S$. Event σ is called *disabled* if σ cannot be executed in $G||S$. It follows from the definition of synchronous composition that $s \in \mathrm{L}(G||S_\mathrm{o})$ if and only if $s \in \mathrm{L}(G)$ and $\mathrm{p_o}(s) \in \mathrm{L}(S_\mathrm{o})$. S_o can only influence observable events. Unobservable events that can be executed in G cannot be disabled by S_o. The synchronous composition is not capable of modeling the interaction between supervisor and uncontrolled system with controllable unobservable events. In this section it will be assumed that all controllable events are observable. In the next section the more general case with no constraints on the controllable and observable event sets will be treated. There, it will be shown how that control problem can be remodeled into the control problem discussed in this section.

Definition 5.3 Let $\mathcal{CO}(G, E, \Sigma_o)$ be the set of all supervisors that solve the supervisory control problem with partial observation.

$$\mathcal{CO}(G, E, \Sigma_\mathrm{o}) = \{S_\mathrm{o} \in \Pi(\Sigma_\mathrm{o}) : G||S_\mathrm{o} \precsim E \text{ and } S_\mathrm{o} \text{ is complete}\}.$$

If $\mathcal{CO}(G, E, \Sigma_\mathrm{o})$ is nonempty then the *supremal supervisor under partial observation*, is the supremal of $\mathcal{CO}(G, E, \Sigma_\mathrm{o})$.

$$\begin{aligned} S_o^{\uparrow} &= \sup_{\precsim} \mathcal{CO}(G, E, \Sigma_o) \\ &= \bigsqcup \{S_o \in \Pi(\Sigma_o) : G||S_o \precsim E \text{ and } S_o \text{ is complete}\}. \end{aligned}$$

If $\mathcal{CO}(G, E, \Sigma_o)$ is empty then $S_o^{\uparrow}$ is empty.

Theorem 5.4 *Let G, $E \in \Pi(\Sigma)$ and $\Sigma_o \subseteq \Sigma$. Let $S_o^{\uparrow}$ be defined as above. Let $S_o \in \Pi(\Sigma_o)$. Then*

$$S_o \precsim S_o^{\uparrow} \iff G||S_o \precsim E \textit{ and } S_o \textit{ is complete.}$$

The proof goes analogous to the proof of Theorem 3.2 and is therefore omitted.

The supremal supervisor under partial observation $S_o^{\uparrow}$ can be computed similar to the computation of $S^{\uparrow}$. Let H_o be defined by

$$\begin{aligned} \Sigma(H_o) &= \Sigma_o, \\ \mathrm{L}(H_o) &= \sup_{\subseteq} \{K \subseteq \Sigma_o^* : K = \overline{K} \text{ and } \mathrm{p}_o^{-\uparrow}(K) \cap \mathrm{L}(G) \subseteq \mathrm{L}(E)\} \\ &= \{s_o \in \Sigma_o^* : \mathrm{p}_o^{-1}(\overline{s_o}) \subseteq (\Sigma^* - \mathrm{L}(G)) \cup \mathrm{L}(E)\}, \\ \mathrm{ref}(H_o, s_o) &= \{R \subseteq \Sigma_o : \forall s \in \mathrm{p}_o^{-1}(s_o) \cap \mathrm{L}(G),\ \forall R_g \in \mathrm{ref}(G, s), \\ &\qquad R_g \cup R \in \mathrm{ref}(E, s) \text{ and } R \cap \Sigma_{uc} \subseteq \rho(\mathrm{L}(G), s)\ \}. \end{aligned}$$

Proposition 5.5 *Let $S_o^{\uparrow}$ be the supremal supervisor under partial observation and let H_o be defined as above. Then $S_o^{\uparrow} = \Pi^{\uparrow}(H_o)$.*

Proof. By Proposition 3.3 and Theorem 5.4 it is sufficient to show that for any $A \in \Pi(\Sigma_o)$

$$A \precsim H_o \iff G||A \precsim E \text{ and } A \text{ is complete.}$$

As A is a process, $\mathrm{L}(A)$ is prefix closed. So

$$\begin{aligned} &\mathrm{L}(A) \subseteq \mathrm{L}(H_o) \\ &\quad\iff \mathrm{L}(A) \subseteq \sup_{\subseteq} \{K \subseteq \Sigma_o^* : K = \overline{K} \text{ and } \mathrm{p}_o^{-\uparrow}(K) \cap \mathrm{L}(G) \subseteq \mathrm{L}(E)\} \\ &\quad\iff \mathrm{p}_o^{-\uparrow}(\mathrm{L}(A)) \cap \mathrm{L}(G) \subseteq \mathrm{L}(E) \\ &\quad\iff \mathrm{L}(G||A) \subseteq \mathrm{L}(E). \end{aligned}$$

For all $s_o \in \mathrm{L}(A)$

$$\begin{aligned} &\mathrm{ref}(A, s_o) \subseteq \mathrm{ref}(H_o, s_o) \\ &\quad\iff \forall R \in \mathrm{ref}(A, s_o),\ \forall s \in \mathrm{p}_o^{-1}(s_o) \cap \mathrm{L}(G),\ \forall R_g \in \mathrm{ref}(G, s), \\ &\qquad\qquad R_g \cup R \in \mathrm{ref}(E, s), \text{ and} \\ &\qquad\qquad R \cap \Sigma_{uc} \subseteq \rho(\mathrm{L}(G), s) \\ &\quad\iff \mathrm{ref}(G||A) \subseteq \mathrm{ref}(E, s), \text{ and} \\ &\qquad A \text{ is complete.} \end{aligned}$$

□

As with the basic supervisory control problem, the supremal supervisor under partial observation is not intended to be implemented directly. It is used as an intermediate result to describe the class of all possible solutions. Usually the deterministic, least restrictive supervisor whose language contains only traces that can be executed by G is a good candidate to be implemented. Just as in Chapter 3 this supervisor is equal to the process $\mathrm{Det}(\mathrm{L}(G||S_o^{\uparrow}))$.

The supervisory control problem with partial observation has also been studied extensively in language based settings [13, 36]. It has been shown that under the condition that all controllable events are observable, there exists a solution if and only if there exists a nonempty controllable language, which satisfies also another condition, called normality. A language $K \subseteq \mathrm{L}(G) \cap \mathrm{L}(E)$ is called normal (w.r.t. $\mathrm{L}(G)$ and Σ_o) if

$$K \;=\; \mathrm{L}(G) \cap \mathrm{p}_o^{-\uparrow}(\mathrm{p}_o(K)).$$

Similar to the basic supervisory control problem it can be shown that the supremal controllable, reducible and normal language K, contained in $\mathrm{L}(G) \cap \mathrm{L}(E)$ is equal to $\mathrm{L}(G||S_o^{\uparrow})$.

5.2. Controllable Unobservable Events

Some discrete event models contain events that can be disabled by the supervisor but that cannot be observed. For instance, a computer system often uses flags. Flags are data-bits that, if they are set, allow certain events to occur, and if they are zero, prevent certain events from occurring. The supervisor usually cannot observe these events, but it can control them by setting or resetting the flags. These events can be considered as controllable and unobservable.

In the framework presented thus far supervisors can only control events that are in their alphabet. The framework relies heavily on this fact. Introduction of controllable unobservable events would require a whole new setup for the framework. A mechanism needs to be introduced to model the enablement and disablement of controllable unobservable events. The synchronous composition is not capable of describing such a mechanism. Instead of adapting the whole framework, we will follow a different approach. The essential aspect of this approach is that the control of the unobservable events will be considered from a different point of view. This different point of view will lead to new insights in the modeling of partially observed discrete event systems.

It will be shown that the control problem with controllable unobservable events can be remodeled such that all controllable events are observable. This implies that a supremal supervisor can then be synthesized. Another motivation for a different modeling approach is given by the following argument.

Consider system G shown in figure 5.1. All events are controllable. Events b and c are observable. The specification E is such that state 6 of G is illegal. If initially event j is enabled then directly after the observation of event b the system can be either in state 3 or 4. Now event c needs to be disabled

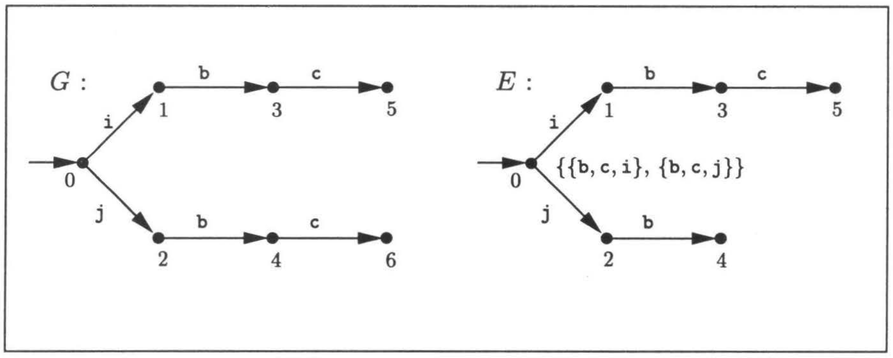


Figure 5.1: System with controllable unobservable events.

to guarantee that the system cannot go to the illegal state 6. On the other hand, if initially event j was disabled then after the observation of event b the system can be only in state 3. Now event c can be safely enabled. We can conclude that the legal control actions after the observation of event b depend on the control actions taken earlier. It can be shown that in general the control possibilities depend not only on the observed trace but also on the controllable unobservable events that were disabled along this trace. In order to describe the control options after the observed trace b it is necessary to distinguish between: 'first enabling event j and then observing b' and 'first disabling j and then observing b'. It is necessary to keep track of the control actions taken with respect to event j along the observed trace in order to be able to model all control possibilities after event b.

The modeling paradigm presented next will be such that the control actions taken along the observed trace are taken into account to describe the control options.

5.3. Flag Events

In the beginning of this section we mentioned the use of flags to implement controllable unobservable events. It was argued that the events controlled by these flags can be considered as controllable and unobservable. Taking another point of view, one can consider the setting and resetting of the flags as controllable and observable events in the system. Unobservable events are enabled only if the corresponding flag is set by one of these flag setting events. The supervisor can influence the unobservable events by enabling and disabling the right flag setting events. There is no need for the supervisor to control the unobservable events directly. The unobservable events can therefore be considered uncontrollable. Modeling flags in this way results in a model in which all controllable events are observable.

After the (re-)modeling, flag events are treated as ordinary controllable, observable events. The synthesis procedure of Section 5.1 can be applied to obtain a supremal supervisor.

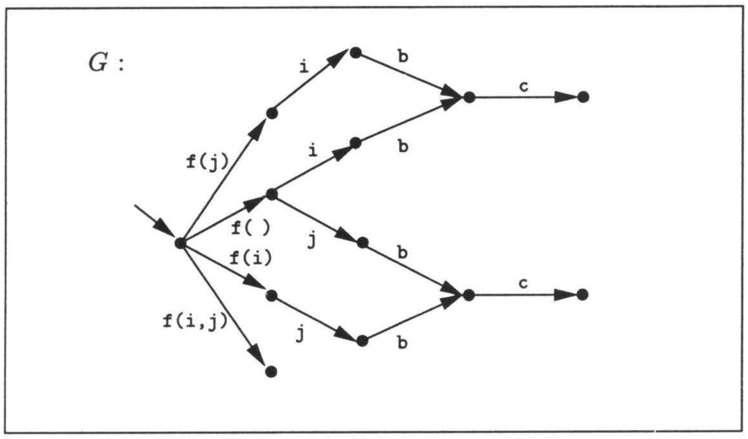


Figure 5.2: Modeling control of unobservable events with flag events.

In order to describe the setting and resetting of the flags we introduce a set of *flag events*, denoted Σ_f, such that each event in this set corresponds with a subset of $\Sigma_c \cap \Sigma_{uo}$. The execution of a flag event means that the corresponding subset of events is disabled. One can regard the flag events as an encoding of the disabled controllable unobservable event set. Figure 5.2 shows system G of Figure 5.1 modeled with flag events. The set of flag events is $\{\mathtt{f()},\mathtt{f(i)},\mathtt{f(j)},\mathtt{f(i,j)}\}$. The events between brackets denote the disabled unobservable events. As the flag events are observable, the supervisor can deduce from the observed trace the control actions taken along this trace. Allowing the supervisor to use information on the past inputs is quite common in other control areas, such as stochastic control and dynamic stochastic games.

The control problem with flag events is to find a complete supervisor $S_o \in \Pi(\Sigma_o \cup \Sigma_f)$ such that $P_e(G||S_o) \precsim E$, where P_e denotes the projection from $\Sigma \cup \Sigma_f$ onto Σ. This control problem can be solved with the methods described in Chapter 3, Chapter 4, and Section 5.1.

In Section 5.1 the set of disabled events is defined to be the set of events that can be executed by G but not by $G||S_o$. One can regard the disabled event sets as inputs to the uncontrolled system. The system can generate only those events that are not in the disabled event set. The supervisor can be regarded as a (nondeterministic) mapping from event sequences generated by the uncontrolled system to the sets of disabled events. The logical relations involved in this interaction are equal to those of the interaction by synchronous composition. Both methods describe the same interaction, but from a different point of view. The use of disabled event sets allows us to talk about control inputs (control actions) and outputs (i.e. traces generated by the uncontrolled system). See Figure 5.3.a. From this point of view it shows that in the original framework the next control action depends only on the outputs of the uncontrolled system and not on the previous control actions.

Flag events can be regarded as both control inputs and as part of the output generated by the uncontrolled system. See Figure 5.3.b. The next control action depends also on the previously executed flag events (control inputs).

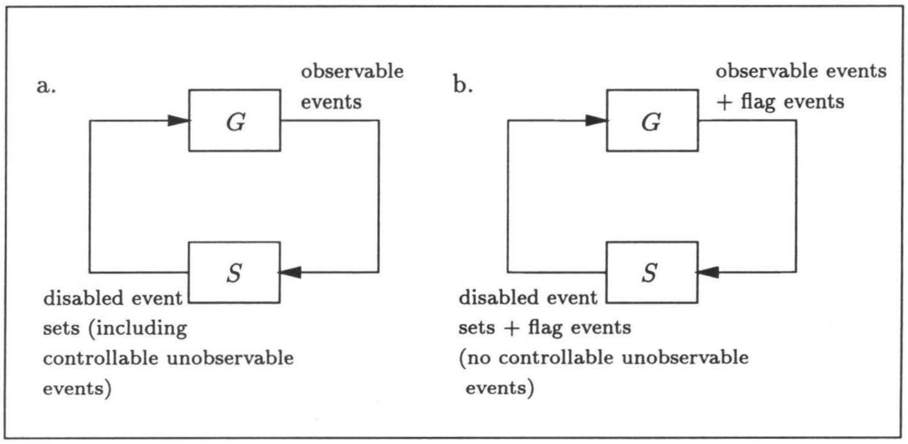

a.
observable events
G
S
disabled event sets (including controllable unobservable events)
b.
observable events + flag events
G
S
disabled event sets + flag events (no controllable unobservable events)

Figure 5.3: Input and output in discrete event control.

In the original control problem formulation with controllable unobservable events the supervisor has the ability to change the set of disabled events after the occurrence of an observable event, and before a next event can occur. One can say that the supervisor enforces the set of disabled events. In the control problem formulation with flag events it is therefore also required that a supervisor can 'enforce' a set of disabled events. This means that the supervisor should be able to execute a flag event before the next event can occur. As the control framework cannot handle forced events, the plant must give the supervisor the opportunity to choose a flag event after each observable event. This is modeled by allowing in the uncontrolled system only flag events after an observable event. Now, the supervisor can 'enforce' a flag event by disabling all the other events. The only event which is then possible in the controlled system is the chosen flag event. Eventually this flag event will be executed. It appears as if the supervisor enforces the flag event. After the flag event is executed, the system continues as usual until the next observable event occurs.

Another method would be to introduce a special class of forced events. A similar method for real-time system is used in [8]. If events from this class of forced events are enabled in the controlled system, then they have priority over the other events. One of the forced events will be executed. Further research is needed to include forced events in the framework.

Even if the controllable unobservable events are not controlled by flag bits, then still the system can be modeled such that all controllable events are observable. For this, model the actions that disable and enable the controllable unobservable events as extra (flag) events. The unobservable events can be controlled indirectly via these extra (flag) events. So, the unobservable events can be considered uncontrollable. All controllable events in the model will be observable.

Note that modeling is part of the control problem. The designer has some freedom in deciding which model to use. Of course, the model has to describe the important characteristics of the real life system. But besides that he or she can freely choose whether to use controllable unobservable events or to use flag events. The consideration given above shows that the use of flag events has some advantages. All controllable events are observable, so we can use the standard synthesis methods presented in Chapter 3, Chapter 4, and Section 5.1. The existence of a unique supremal supervisor is guaranteed (although this supremal may be empty).

This last point might seem strange. How is it possible that in the original control problem with controllable unobservable events in general no unique supremal solution exists [36], but the same control problem modeled with flag events has a supremal solution? This will be discussed in the next section.

5.4. Supremal Supervisor and Control Inputs

Consider again system G given in Figure 5.2 and the specification E given in Figure 5.1. Using the synthesis method described in Section 5.1 the supremal supervisor under partial observation can be computed. It is shown in Figure 5.4.

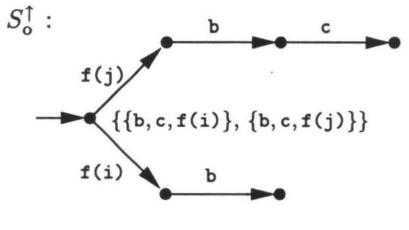


Figure 5.4: Supremal supervisor.

Initially the supervisor must either enable i and disable j, or disable i and enable j. In the original framework these two alternatives could not be described in one process. Using the flag events gives us the possibility to include both options in the supremal supervisor. The actual control action will depend on the flag event that is executed.

Flag events can be regarded as inputs to the uncontrolled system. This is the crucial difference between the approach with controllable unobservable events and the approach with flag events. With flag events the control actions of the supervisor can depend on (part of) the previously taken control. It is quite common in various fields of control theory to allow control laws to depend on the past control inputs.

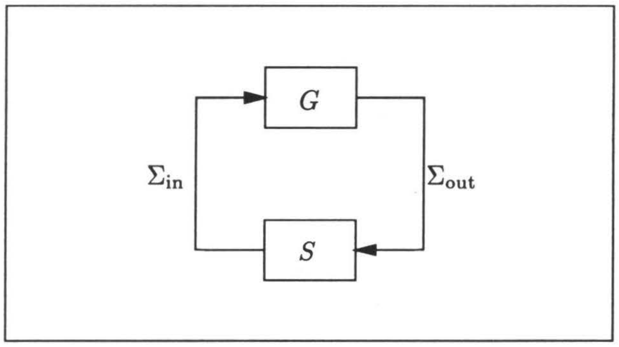


Figure 5.5: Input-output automata approach

To extend this line of thinking, all control actions of the supervisor can be modeled by input events. This will result in an input-output model based approach [4, 24]. See Figure 5.5. Given are a set of input events Σ_{in} and a set of output events Σ_{out}. It is assumed that the input events are under control of the supervisor, S. The output events are under control of the uncontrolled system, G. System G generates output events according to the input events it receives from supervisor S. The behavior of G is described as a language $\mathrm{L}(G) \subseteq (\Sigma_{\text{in}} \cup \Sigma_{\text{out}})^*$. Considering the graphical image shown in Figure 5.5, one might think that S uses only the output events of G to choose a next input event. This is however not true. The input events are combined inside S with the output events. In this way the generated trace $s \in \mathrm{L}(G) \subseteq (\Sigma_{\text{in}} \cup \Sigma_{\text{out}})^*$ is obtained. The supervisor maps this trace to the next input event. So the next input event depends also on the input events in trace s.

The control law of a supervisor modeled as an input-output automaton depends on its previously taken control actions. Therefore the control problem with partial observation will have a supremal solution (although it might be empty). To our knowledge, this property of input-output automata has not been stated yet.

In [4, 24] the authors claim that modeling with input-output automata is closer related to real life problems. This is their main motivation for the use of input-output automata. Synthesis methods are known for control problems based on input-output automata.

In [27] K. Inan describes a supremal supervisor for the supervisory control problem with partial observation in which the control options are modeled by a nondeterministic choice. Our supremal solution can be converted to the model Inan uses by changing all flag events into internal events. A choice between flag events then becomes a nondeterministic choice. We prefer the method described in this paper because the flag events express more clearly the control options of the supervisor. Also, introducing flag events changes the control problem with controllable unobservable events to a control problem in which all controllable events are observable. This control problem can be solved with the existing synthesis methods.

5.5. Arbitrary Controllable, Observable, and External Event Sets

In Chapter 3 the set of controllable events was introduced. In Chapter 4 the control problem with partial specification was solved. In that control problem the set of external events Σ_e is contained strictly in $\Sigma(G)$. In this chapter the control problem with partial observation is discussed. The set of observable events $\Sigma_o = \Sigma(S)$ is defined. Next, we will show how the different parts relate to each other. We will consider arbitrary controllable, observable, and external event sets, and show how control problems with arbitrary event sets can be solved.

It makes no sense to consider events that are not in the alphabet of G. System G describes all sequences of events that are possible and relevant for the control problem. If an event is controllable, observable, or external then it is relevant for the control problem. So G should describe in which sequences the event can occur. We can restrict our attention to situations with

$$\Sigma_c \cup \Sigma_o \cup \Sigma_e \subseteq \Sigma(G).$$

If the inclusion relation is strict then there are events in $\Sigma(G)$ that are neither controllable, observable, nor used in the specification. These events are irrelevant for the control problem and can be projected out of the uncontrolled system. The projection of G onto the event set $\Sigma_c \cup \Sigma_o \cup \Sigma_e$ can be considered as uncontrolled system. The control problem thus obtained satisfies

$$\Sigma_c \cup \Sigma_o \cup \Sigma_e = \Sigma(G).$$

If $\Sigma_c \subsetneqq \Sigma_o$ then use the method described in Section 5.2 to remodel the control problem such that $\Sigma_c \subseteq \Sigma_o$. After remodeling the situation is characterized by

$$\Sigma_c \subseteq \Sigma_o, \qquad \Sigma_o \cup \Sigma_e = \Sigma(G).$$

If the specification is partial, i.e. $\Sigma_e \subsetneqq \Sigma(G)$, then the control problem can be reduced to a control problem with full specification using the method described in Chapter 4. After the reduction the control problem satisfies

$$\Sigma_{\mathrm{c}} \subseteq \Sigma_{\mathrm{o}} \subseteq \Sigma_{\mathrm{e}} = \Sigma(G).$$

Now, we have arrived at the situation of a 'standard' control problem with partial observation. In Section 5.1 it is described how this control problem can be solved.

Chapter 6
Decentralized Control

In this chapter fundamental properties of decentralized control problems for discrete event systems are discussed. These control problems arise very naturally from protocol design problems for computer and communication networks.

The following control problem will be discussed. Given a system describing all physically possible behavior, and a specification describing the required behavior, find a pair of supervisors, each observing only part of the system and each controlling only part of system, such that the system controlled by the supervisors satisfies the specification.

First this decentralized supervisory control problem will be reduced such that only those details remain that are important for the analysis of the decentralized nature of the problem. Next, some results from the literature will be discussed. Finally a characterization of the maximal solutions for the decentralized control problem will be presented.

6.1. Simple Framework

In the first chapters of the thesis a framework is introduced to discuss the nondeterministic properties of supervisory control. In this chapter decentralized control will be discussed. A simple framework will be introduced that allows us to concentrate on the decentralized aspects of the control problem.

Throughout this chapter let the global set of events Σ, the global set of controllable events Σ_c, the uncontrolled system G, and the specification E be given.

Definition 6.1 A *supervisor* or *discrete event controller* is defined by a triple

$$S = (\Sigma(S),\ \Sigma_c(S),\ \gamma(S)),$$

where

$$\begin{array}{lcl}\Sigma(S) & \subseteq & \Sigma, \\ \Sigma_c(S) & \subseteq & \Sigma(S), \\ \gamma(S) & : & p_s(L(G)) \to 2^{\Sigma_c(S)},\end{array}$$

and p_s is the projection from Σ to $\Sigma(S)$.

Define the controlled language of supervisor S with respect to G or, for short, the *language* of S as

$$L(S/G) \quad = \quad \{s \in L(G) : \forall v\sigma \in \overline{s}, \sigma \notin \gamma(S, p_s(v))\}.$$

Note that $L(S/G) \subseteq \Sigma^*$.

Let $\mathcal{C}(\Sigma_a)$ denote the set of all supervisors S with event set $\Sigma(S) = \Sigma_a$ and controllable event set $\Sigma_c(S) = \Sigma_c \cap \Sigma_a$. The function $\gamma(S)$ will be called the *control law* of supervisor S. Note that $\gamma(S, s)$ is defined for all $s \in p_s(L(G))$.

The control law $\gamma(S)$ maps each trace $s \in p_s(L(G))$ onto the set of disabled events. In the literature often the set of enabled events is specified [52]. Both approaches are equivalent. Which option one uses is a matter of personal taste.

In the definition above the set of controllable events is taken to be contained in the set of events observable by the supervisor. In general it is possible that a supervisor can influence events it cannot observe. In Section 5.2 it is shown how in this situation a control problem can be remodeled such that all controllable events are observable.

We want to concentrate on the decentralized aspects of the supervisory control problem. We will use a simple framework in which only details relevant for the decentralized nature of the problem are taken into account. Marking, nondeterminism, or failure semantics will not be considered. The argument that we want a simple framework is also the reason that we consider the situation with only two supervisors. We are confident that in the future the results can be extended to more general frameworks and more supervisors.

The basic supervisory control problem needs to be redefined for the new framework. Note that supervisors, as stated in Definition 6.1, can disable only controllable events. So they are always complete. It is not necessary to add a completeness requirement as is done in Chapter 2.

Definition 6.2 The *Basic Supervisory Control Problem* (BSCP) is to find a supervisor S, such that $L(S/G) \subseteq L(E)$.

Ramadge and Wonham showed that there exists a unique supremal solution to this control problem. This supremal can be effectively computed [52]. It is characterized by a language called the supremal controllable sublanguage contained in $\mathrm{L}(G) \cap \mathrm{L}(E)$. As the notion of controllability will not be used any further, we refer the interested reader to the given reference or to Chapter 3 for more information. The only aspect of controllability that will be used in this chapter is that the supremal controllable language can be effectively computed.

Definition 6.3 Let $K^{\uparrow}$ be the supremal controllable sublanguage contained in $\mathrm{L}(G) \cap \mathrm{L}(E)$. The *supremal supervisor*, denoted by $S^{\uparrow}$, is defined by

$$\gamma(S^{\uparrow}, s) = \begin{cases} \{\sigma \in \Sigma_{\mathrm{c}} : s\sigma \in \mathrm{L}(G) \text{ and } s\sigma \notin K^{\uparrow}\}, & \text{if } s \in K^{\uparrow}, \\ \emptyset, & \text{otherwise.} \end{cases}$$

It is not difficult to show that $\mathrm{L}(S^{\uparrow}/G) = K^{\uparrow}$. As $S^{\uparrow}$ is supremal it holds for all supervisors S which solve the given BSCP, that $\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$.

In this paper it will be assumed that BSCP is already solved and that the supremal supervisor $S^{\uparrow}$ is given. It is sufficient to find a supervisor that implements $S^{\uparrow}$, with respect to the implementation relation defined below. Proposition 6.7 shows that this is a valid approach. A supervisor implements the supremal supervisor if and only if the supervisor solves the basic supervisory control problem.

Definition 6.4 Let $S_{\mathrm{a}}, S_{\mathrm{b}}$ be two supervisors such that $\Sigma(S_{\mathrm{a}}) = \Sigma(S_{\mathrm{b}})$. Supervisor S_{a} *implements* S_{b}, denoted by $S_{\mathrm{a}} \sqsubseteq S_{\mathrm{b}}$, if

$$\gamma(S_{\mathrm{b}}, s) \subseteq \gamma(S_{\mathrm{a}}, s) \quad \text{for all } s \in \mathrm{p}(\mathrm{L}(S_{\mathrm{a}}/G)),$$

where p is the projection on $\Sigma(S_{\mathrm{a}}) = \Sigma(S_{\mathrm{b}})$.

Supervisor S_{a} implements S_{b} if it disables at least as much as S_{b}.

Lemma 6.5 *Let $S_{\mathrm{a}}, S_{\mathrm{b}}$ be two supervisors such that $\Sigma(S_{\mathrm{a}}) = \Sigma(S_{\mathrm{b}})$.*

$$S_{\mathrm{a}} \sqsubseteq S_{\mathrm{b}} \Rightarrow \mathrm{L}(S_{\mathrm{a}}/G) \subseteq \mathrm{L}(S_{\mathrm{b}}/G).$$

Proof. This Lemma will be proven by complete induction. The initial step is satisfied as $\varepsilon \in \mathrm{L}(S_{\mathrm{a}}/G)$ and $\varepsilon \in \mathrm{L}(S_{\mathrm{b}}/G)$. For the inductive step let $s \in \mathrm{L}(S_{\mathrm{a}}/G)$ and $s \in \mathrm{L}(S_{\mathrm{b}}/G)$. It will be proven that if $s\sigma \in \mathrm{L}(S_{\mathrm{a}}/G)$ then $s\sigma \in \mathrm{L}(S_{\mathrm{b}}/G)$.

$s\sigma \in \mathrm{L}(S_{\mathrm{a}}/G)$

$\Rightarrow$ $\qquad$ [Definition of $\mathrm{L}(S_{\mathrm{a}}/G)$]

$\qquad s\sigma \in \mathrm{L}(G),\ \sigma \notin \gamma(S_{\mathrm{a}}, s)$

$\Rightarrow$ $\qquad$ [$\gamma(S_{\mathrm{b}}, s) \subseteq \gamma(S_{\mathrm{a}}, s)$]

$\qquad s\sigma \in \mathrm{L}(G),\ \sigma \notin \gamma(S_{\mathrm{b}}, s)$

$\Rightarrow$ $\qquad$ [$s \in \mathrm{L}(S_{\mathrm{b}}/G)$, Definition of $\mathrm{L}(S_{\mathrm{b}}/G)$]

$\qquad s\sigma \in \mathrm{L}(S_{\mathrm{b}}/G)$.

It can be concluded that $\mathrm{L}(S_{\mathrm{a}}/G) \subseteq \mathrm{L}(S_{\mathrm{b}}/G)$. □

The following example will show why the converse of Lemma 6.5 does not hold.

Example 6.6 Let G be the system such that $\mathrm{L}(G) = \{\varepsilon, \mathrm{a}\}$. Define S_{a} by $\gamma(S_{\mathrm{a}}, \varepsilon) = \emptyset$ and $\gamma(S_{\mathrm{a}}, \mathrm{a}) = \emptyset$. Define S_{b} by $\gamma(S_{\mathrm{b}}, \varepsilon) = \emptyset$ and $\gamma(S_{\mathrm{a}}, \mathrm{a}) = \{\mathrm{a}\}$. Then $\mathrm{L}(S_{\mathrm{a}}/G) = \{\varepsilon, \mathrm{a}\} = \mathrm{L}(S_{\mathrm{b}}/G)$, but $\gamma(S_{\mathrm{b}}, \mathrm{a}) \not\subseteq \gamma(S_{\mathrm{a}}, \mathrm{a})$. So $S_{\mathrm{a}} \not\sqsubseteq S_{\mathrm{b}}$.

In Theorem 3.2 it was shown that in the failure semantics based framework a supervisors solves the basic supervisory control problem if and only if it implements the supremal supervisor. Proposition 6.7 states the same result for the framework used in this chapter.

Proposition 6.7 *Let $S^{\uparrow}$ be the supremal supervisor of BSCP.*

$$\begin{aligned} \forall S \in \mathcal{C}(\Sigma), \quad S \sqsubseteq S^{\uparrow} \quad &\iff \quad \mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G) \\ &\iff \quad \mathrm{L}(S/G) \subseteq \mathrm{L}(E). \end{aligned}$$

Proof. ($S \sqsubseteq S^{\uparrow} \Rightarrow \mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$.) This follows directly from Lemma 6.5.

($\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G) \Rightarrow \mathrm{L}(S/G) \subseteq \mathrm{L}(E)$.) This follows directly from the fact that $\mathrm{L}(S^{\uparrow}/G) \subseteq \mathrm{L}(E)$.

($\mathrm{L}(S/G) \subseteq \mathrm{L}(E) \Rightarrow \mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$. This follows from the supremality of $S^{\uparrow}$. See Definition 6.3.

($\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G) \Rightarrow S \sqsubseteq S^{\uparrow}$.) This point will be proven by contradiction. Let $\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$. Let $s \in \mathrm{L}(S/G)$, so also $s \in \mathrm{L}(S^{\uparrow}/G)$. Suppose $\sigma \in \gamma(S^{\uparrow}, s)$ but $\sigma \notin \gamma(S, s)$.

$s \in \mathrm{L}(S/G),\ s \in \mathrm{L}(S^{\uparrow}/G),\ \sigma \in \gamma(S^{\uparrow}, s),\ \sigma \notin \gamma(S, s)$

$\Rightarrow$ $\qquad$ [Definition of $\gamma(S^{\uparrow}, s)$]

$\qquad s \in \mathrm{L}(S/G),\ s\sigma \in \mathrm{L}(G),\ s\sigma \notin \mathrm{L}(S^{\uparrow}/G),\ \sigma \notin \gamma(S, s)$

$\Rightarrow$ $\qquad$ [Definition of $\mathrm{L}(S/G)$]

$\qquad s\sigma \notin \mathrm{L}(S^{\uparrow}/G),\ s\sigma \in \mathrm{L}(S/G)$.

But this contradicts the assumption that $\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$. Hence we can conclude that $\forall s \in \mathrm{L}(S/G)$, $\gamma(S^{\uparrow}, s) \subseteq \gamma(S, s)$. So $S \sqsubseteq S^{\uparrow}$. □

In the rest of this paper we will consider control problems that place extra constraints on the supervisor besides the ones given in BSCP. Proposition 6.7 states that we can first solve BSCP to get the supremal supervisor $S^{\uparrow}$. Next we can look for supervisors that satisfy the extra constraints and that implement $S^{\uparrow}$. In this last step we can concentrate on the extra requirement. As we are mainly interested in the extra requirements imposed by the decentralized nature of the control problem, we will assume that the first step is already solved and that the supremal supervisor $S^{\uparrow}$ is given.

Definition 6.8 The *Basic Supervisory Synthesis Problem* (BSSP) is to find a supervisor $S \in \mathcal{C}(\Sigma(S^{\uparrow}))$ such that $S \sqsubseteq S^{\uparrow}$.

Often in the literature supervisors are defined as languages instead of control maps. We choose to use control maps as it allows us to divide the control problem into two steps. In the first step the supremal supervisor is synthesized. In this step the controllability condition plays an important role. In the second step we can concentrate on the decentralized aspect of the control problem. Proposition 6.7 shows that we do not have to consider the controllability condition in this step. If supervisors are defined as languages, then also the problem can be divided into two parts. The synthesis problem of the second part is then defined as follows: find a supervisor S such that $\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$ and $\mathrm{L}(S/G)$ is controllable. It is necessary to check for controllability, as $\mathrm{L}(S/G) \subseteq \mathrm{L}(S^{\uparrow}/G)$ does not imply that $\mathrm{L}(S/G)$ is controllable. So in the second step we still have to consider controllability. Using control maps we can forget about controllability in the second step and concentrate on the decentralized aspects of the control problem. It is not too difficult to adapt the results in this chapter to a language based approach.

6.2. Decentralized Supervisory Synthesis Problem

Up till now we have only looked at supervisors that can observe the whole event set and that control G by themselves. Now we will look at the decentralized control problem where we have two supervisors, each observing a part of the event set, and each controlling only a part of the system. See Figure 6.1. The two supervisors together have to control G such that the language of the controlled system is contained in the language of E. Note that the specification is given for the whole controllable system. This is usually referred to as a global specification [53, 55]. If the specification can be decomposed into two local specifications, one for each supervisor, then the decentralized control problem can be reduced to two independent supervisory control problems. In each of these local control problems a single supervisor is synthesized. This control problem has already been solved by F. Lin and M. W. Wonham [37].

In the sequel we will assume that the specification is global and cannot be decomposed into local specifications.

As stated before we will assume BSCP is already solved and the supremal supervisor $S^{\uparrow}$ is known. By Proposition 6.7 it is sufficient to find a decentralized implementation of $S^{\uparrow}$ to solve the decentralized supervisory control problem.

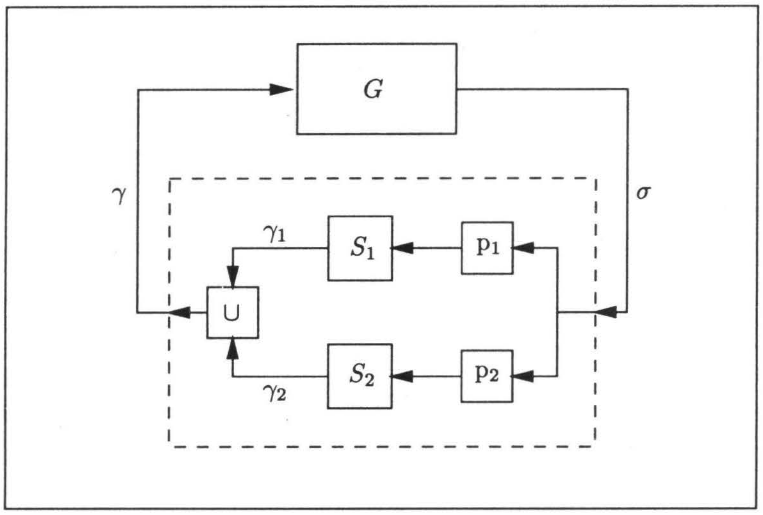


Figure 6.1: The decentralized supervisory control problem.

First it will be defined how two decentralized supervisors co-operate. An event is disabled by the combination of the two supervisors if it is disabled by at least one of them.

Definition 6.9 Let S_1 and S_2 be two supervisors. The *composition* of S_1 and S_2 is denoted $S_1 \wedge S_2$, and defined by

$$\begin{aligned}
\Sigma(S_1 \wedge S_2) &= \Sigma_1 \cup \Sigma_2, \\
\Sigma_c(S_1 \wedge S_2) &= \Sigma_c(S_1) \cup \Sigma_c(S_2). \\
\gamma(S_1 \wedge S_2, s) &= \gamma(S_1, \mathrm{p}_1(s)) \cup \gamma(S_2, \mathrm{p}_2(s)) \qquad \text{for all } s \in \mathrm{p}_{1,2}(\mathrm{L}(G)),
\end{aligned}$$

where p_1 denotes the projection on $\Sigma(S_1)$, p_2 denotes the projection on $\Sigma(S_2)$, and $\mathrm{p}_{1,2}$ denotes the projection on $\Sigma(S_1 \wedge S_2)$.

Proposition 6.10 $\quad \mathrm{L}(S_1 \wedge S_2/G) = \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G).$

Proof. The reasoning follows from Definition 6.1.

$$\begin{aligned}
& s \in \mathrm{L}(S_1 \wedge S_2/G) \\
& \quad \iff \quad s \in \mathrm{L}(G),\ \forall v\sigma \in \overline{s},\ \sigma \notin \gamma(S_1 \wedge S_2, \mathrm{p}_{1,2}(v)) \\
& \quad \iff \quad s \in \mathrm{L}(G),\ \forall v\sigma \in \overline{s},\ \sigma \notin \gamma(S_1, \mathrm{p}_1(v)),\ \sigma \notin \gamma(S_2, \mathrm{p}_2(v)) \\
& \quad \iff \quad s \in \mathrm{L}(S_1/G),\ s \in \mathrm{L}(S_2/G) \\
& \quad \iff \quad s \in \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G).
\end{aligned}$$

Definition 6.11 Let the supremal supervisor $S^{\uparrow}$ be given. Let $\Sigma_1, \Sigma_2 \subseteq \Sigma(S^{\uparrow})$ be two event sets such that $\Sigma_1 \cup \Sigma_2 = \Sigma(S^{\uparrow})$. The *Decentralized Supervisory Synthesis Problem* (DSSP) is to find a pair of supervisors $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ such that

$$S_1 \wedge S_2 \quad \sqsubseteq \quad S^{\uparrow}.$$

In this definition we made two important assumptions. The one is that $\Sigma_1 \cup \Sigma_2 = \Sigma(S^{\uparrow})$. The other is that, according to the definition of $\mathcal{C}(\Sigma_i)$, the set of controllable events of supervisor S_i, $\Sigma_{c,i}$, is equal to $\Sigma_i \cap \Sigma_c$ for $i = 1, 2$.

Consider the case where $\Sigma_1 \cup \Sigma_2 \subsetneq \Sigma(S^{\uparrow})$. If $\Sigma_c \subseteq \Sigma_1 \cup \Sigma_2$ then we can compute the supremal supervisor under partial observation, with observation alphabet $\Sigma_1 \cup \Sigma_2$. See [13, 36] and Section 5.1. Equivalently to Proposition 6.7, it can be shown that a supervisor implements this supremal supervisor if and only if it solves the control problem under partial observation. We can assume that this control problem is already solved and that the supremal supervisor under partial observation is given. So this control problem can be reduced to DSSP.

If $\Sigma_c \not\subseteq \Sigma_1 \cup \Sigma_2$ then the control problem can be remodeled in such a way that all controllable events are observable. See Section 5.2.

The other assumption is that $\Sigma_{c,i} = \Sigma_i \cap \Sigma_c$, $i = 1, 2$. That is, the controllable events of supervisor S_i are observable by S_i, and an event that is controllable by $S^{\uparrow}$ and observable by S_i is also controllable by S_i. This is the same constraint as given by Rudie [53, 55] under which decomposability of the closed loop language is necessary and sufficient for the existence of a decentralized solution. It is argued that in most communication problems these constraints are satisfied. Again, as we want to keep the model simple, we do not consider systems that fail to satisfy this constraint. We hope that in the future these constraints can be relaxed.

6.3. Optimal and Maximal Solutions

Traditionally in Discrete Event Control, supervisors are looked for that restrict the uncontrolled system as little as possible. A solution is considered optimal if the language of the system controlled by this optimal supervisor is larger than the languages of all other solutions.

Definition 6.12 A pair of supervisors $(S_1^{\uparrow}, S_2^{\uparrow}) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is called an *optimal decentralized solution* if it is a solution, i.e.

$$S_1^\uparrow \wedge S_2^\uparrow \ \sqsubseteq \ S^\uparrow,$$

and for all pairs $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$

$$S_1 \wedge S_2 \sqsubseteq S^\uparrow \quad \Rightarrow \quad \mathrm{L}(S_1 \wedge S_2/G) \subseteq \mathrm{L}(S_1^\uparrow \wedge S_2^\uparrow/G).$$

Recall from [53, 55] the definition of decomposability. A language $K \subseteq \mathrm{L}(G)$ is called decomposable if

$$K \ = \ \mathrm{p}_1^{-\uparrow}(\mathrm{p}_1(K)) \cap \mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K)) \cap \mathrm{L}(G).$$

Rudie showed that, under the given assumptions, there exists a decentralized solution, $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$, such that the language of the controlled system, $\mathrm{L}(S_1 \wedge S_2/G)$, is equal to a given language $K \subseteq \mathrm{L}(G)$ if and only if K is decomposable. The set of decomposable languages is not closed under arbitrary unions. It is therefore not guaranteed that this set contains a unique supremal element. This implies that in general the optimal decentralized solution does not exist. There may exist several, mutually incomparable, maximal solutions.

Definition 6.13 A pair of supervisors $(S_1^\square, S_2^\square) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is called a *maximal decentralized solution* if it is a solution, i.e.

$$S_1^\square \wedge S_2^\square \ \sqsubseteq \ S^\uparrow,$$

and there does not exist a pair $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ such that

$$S_1 \wedge S_2 \sqsubseteq S^\uparrow \quad \text{and} \quad \mathrm{L}(S_1^\square \wedge S_2^\square/G) \subsetneq \mathrm{L}(S_1 \wedge S_2/G).$$

The set of decomposable languages is closed under arbitrary intersections. It therefore contains a unique infimal element. Rudie posed the following control problem. Given lower bound $\mathrm{L}(A) \subseteq \Sigma^*$ and upper bound $\mathrm{L}(E) \subseteq \Sigma^*$, find a pair $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$, such that

$$\mathrm{L}(A) \ \subseteq \ \mathrm{L}(S_1 \wedge S_2/G) \ \subseteq \ \mathrm{L}(E).$$

She showed there exists a solution to this control problem if and only if the infimal decomposable language containing $\mathrm{L}(A)$ is contained in $\mathrm{L}(E)$. Although, this infimal is useful to solve the existence question, it often does not give a satisfactory solution. The following example shows that it is in general not trivial how to define the lower bound $\mathrm{L}(A)$.

Example 6.14 Consider the alternating bit protocol [53, 55, 57]. This protocol achieves the reliable transmission of messages across an unreliable connection. To achieve this, the sender attaches to each message an extra bit containing either a zero or a one. The protocol can start with either a zero or a one attached to the first message. Consequently either the message with a one or a zero attached is disabled initially. If the lower bound allows a zero attached to the first message then the protocol cannot disable this message. It cannot choose the option where a one is attached to the first message. The lower bound $\mathrm{L}(A)$ should allow for both options. Therefore it cannot contain either of the options as this would exclude the other option. The only lower bound that allows both options is the empty language. Unfortunately the infimal decomposable language derived from the empty language does not give a satisfactory solution. See also Section 2.5.

Another suggestion presented in [53, 55] was to look for the suboptimal solution characterized by the strong decomposability condition. A language $K \subseteq \mathrm{L}(G)$ is called *strongly decomposable* (w.r.t. Σ_1 and Σ_2) if

$$K = (\mathrm{p}_1^{-\uparrow}(\mathrm{p}_1(K)) \cup \mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K))) \cap \mathrm{L}(G).$$

This condition is closed under arbitrary unions. So the supremal strongly decomposable language exists. Recall from [36] the definition of normality. A language $K \subseteq \mathrm{L}(G)$ is called *normal* (w.r.t. $\Sigma_o \subseteq \Sigma$) if

$$K = \mathrm{p}_o^{-\uparrow}(\mathrm{p}_o(K)) \cap \mathrm{L}(G).$$

Normality of a language K is a sufficient condition for the existence of supervisor that can observe events in Σ_o, and that achieves K as language of the controlled system.

Proposition 6.15 *If $K \subseteq \mathrm{L}(G)$ is strongly decomposable w.r.t. Σ_1 and Σ_2 then K is normal w.r.t. Σ_1 and normal w.r.t. Σ_2.*

Proof. The inclusion $K \subseteq \mathrm{p}_i^{-\uparrow}(\mathrm{p}_i(K)) \cap \mathrm{L}(G)$ is satisfied for all languages contained in $\mathrm{L}(G)$. So, it is sufficient to prove $K \supseteq \mathrm{p}_i^{-\uparrow}(\mathrm{p}_i(K)) \cap \mathrm{L}(G)$. By the definition of strongly decomposability

$$\begin{aligned} K &= (\mathrm{p}_1^{-\uparrow}(\mathrm{p}_1(K)) \cup \mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K))) \cap \mathrm{L}(G) \\ &= (\mathrm{p}_1^{-\uparrow}(\mathrm{p}_1(K)) \cap \mathrm{L}(G)) \cup (\mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K)) \cap \mathrm{L}(G)) \\ &\supseteq \mathrm{p}_i^{-\uparrow}(\mathrm{p}_i(K)) \cap \mathrm{L}(G), \text{ for } i = 1, 2. \end{aligned}$$ □

The consequence of this proposition is that, if language K is strongly decomposable, then one supervisor, either $S_1 \in \mathcal{C}(\Sigma_1)$ or $S_2 \in \mathcal{C}(\Sigma_2)$, can obtain K as language of the controlled system. The other supervisor is not needed. Obviously, strong decomposability is too strong a restriction for decentralized control problems.

It can be concluded that the existing results for decentralized supervisory control problems do not satisfy the needs from control engineering.

Maximal Solutions

In this chapter a characterization of maximal solutions for decentralized control problems will be derived. Is it useful to look for maximal solutions? If a solution is maximal then this does not imply that it is a good solution. For instance, a maximal solution may allow a lot of unimportant traces and disable all important ones. Another solution which allows less unimportant traces but more important ones may be considered a better solution. However, we believe there are some good reason to investigate the characteristics of maximal solutions. The first and most important reason is that it gives us valuable insight in the fundamental properties of decentralized control problems. This insight may be used to derive algorithms that can synthesize 'good' (in whatever sense) solutions, whether they are maximal or not.

Another reason why we believe maximality is important, is that these 'good' solutions will probably be maximal. So, although maximality of a solution does not imply that this solution is useful, a solution that is useful (good in some sense) will most likely be maximal. If a characterization of all maximal solutions can be given, then all 'good' solutions will satisfy this characterization. So this characterization limits the class of solutions in which the good ones can be found.

Suppose a solution is given, but it is not fully satisfactory. One can ask the question whether the solution can be extended to obtain a better one. This is possible only if the given solution is not yet maximal. So also in this case a characterization of the maximal solutions will be useful.

6.4. Projection of the Supremal Supervisor

In [30] Kozak proposes projections of the supremal supervisor as a solution to the decentralized control or synthesis problem.

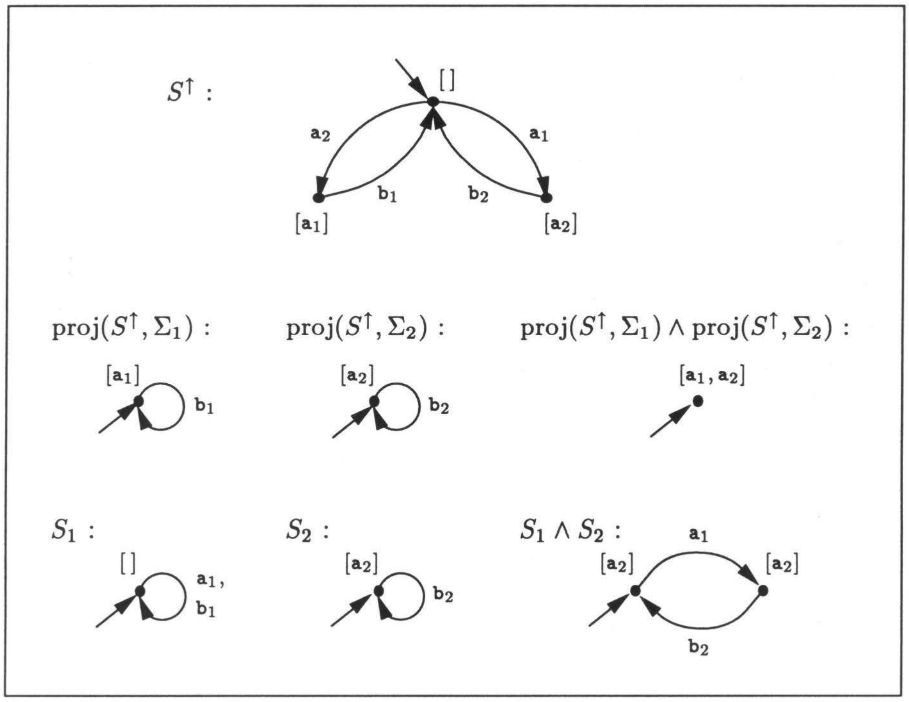
Definition 6.16 The *projection of the supremal supervisor* to event set $\Sigma_a \subseteq \Sigma(S^\uparrow)$ is denoted by $\mathrm{proj}(S^\uparrow, \Sigma_a)$. It is defined for all $s_a \in \mathrm{p_a}(\mathrm{L}(G))$ by

$$\gamma(\mathrm{proj}(S^\uparrow, \Sigma_a), s_a) = \{\sigma \in \Sigma_c(S_a) : \exists s \in \mathrm{p}_a^{-1}(s_a) \cap \mathrm{L}(S^\uparrow/G) \text{ s.t. } \sigma \in \gamma(S^\uparrow, s)\}.$$

Proposition 6.17 ([30], Lemma 5.1)

$$\mathrm{proj}(S^\uparrow, \Sigma_1) \wedge \mathrm{proj}(S^\uparrow, \Sigma_2) \sqsubseteq S^\uparrow.$$

Kozak calls $\mathrm{proj}(S^\uparrow, \Sigma_1) \wedge \mathrm{proj}(S^\uparrow, \Sigma_2)$ the *fully decentralized solution*. In general the infimal decomposable solution of Rudie and the projected solution of Kozak are incomparable. However, if the given lower bound, $\mathrm{L}(A)$, is the empty trace, then the projected solution is larger then the infimal decomposable solution. But, even if $\Sigma_1 \cap \Sigma_2 = \emptyset$, the fully decentralized solution is in general not maximal. Consider the following example.



$S^{\uparrow}$:
[]
a2
a1
b1
b2
[a1]
[a2]
$\text{proj}(S^{\uparrow}, \Sigma_1)$:
[a1]
b1
$\text{proj}(S^{\uparrow}, \Sigma_2)$:
[a2]
b2
$\text{proj}(S^{\uparrow}, \Sigma_1) \wedge \text{proj}(S^{\uparrow}, \Sigma_2)$:
[a1, a2]
S_1 :
[]
a1, b1
S_2 :
[a2]
b2
$S_1 \wedge S_2$:
a1
[a2]
[a2]
b2

Figure 6.2: The fully decentralized solution is in general not maximal.

Example 6.18 Consider the supremal supervisor and the fully decentralized solution given in Figure 6.2. In this example $\Sigma_1 = \{\mathtt{a}_1, \mathtt{b}_1\}$, $\Sigma_2 = \{\mathtt{a}_2, \mathtt{b}_2\}$, and $\Sigma_c = \{\mathtt{a}_1, \mathtt{a}_2\}$. The pair $(\mathrm{proj}(S^\dagger, \Sigma_1), \mathrm{proj}(S^\dagger, \Sigma_2))$ is not maximal, because the pair (S_1, S_2) results in a strictly larger controlled language.

Supervisor $\mathrm{proj}(S^\dagger, \Sigma_1)$ disables event $\mathtt{a}_1$ because the uncontrolled system can execute event $\mathtt{a}_2$, after which event $\mathtt{a}_1$ must be disabled. However, as supervisor S_2 disables $\mathtt{a}_2$ it is not necessary for supervisor S_1 to disable $\mathtt{a}_1$. The pair of supervisors obtained by projection from the supremal supervisor is in general not maximal because the supervisors only take into account the control actions of the supremal supervisor. They do not consider the control law of the other supervisor. In order to obtain a maximal solution it is necessary that the supervisors take into account the control law of the other supervisor. So, to synthesize supervisor S_1 one should already know the control law of supervisor S_2, and to synthesize S_2 one should already know the control law of supervisor S_1. It is this cyclic dependency that makes the synthesis of decentralized controllers such a hard problem.

6.5. Nash and Strong Nash Equilibria

Decentralized stochastic control has been studied extensively. It is related to game and team theory. See [6, 22, 41, 49]. In these fields of research a so called cost function is used. This cost function maps a decentralized solution to a real number. A solution is considered optimal if it has the smallest cost. Using cost functions, all solutions can be compared with each other. In the field of decentralized supervisory control, solutions are compared by the language of the controlled system. This ordering is not complete. Some solutions may not be comparable.

In game and team theory the notion of Nash equilibrium plays an important role. It will be shown that Nash equilibria are also important for decentralized supervisory control . A pair of supervisors forms a Nash equilibrium if each supervisor cannot improve the controlled language when the other remains the same.

Definition 6.19 A pair of supervisors $(S_1^\circ, S_2^\circ) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is called a *Nash equilibrium* if it is a solution, i.e.

$$S_1^\circ \wedge S_2^\circ \sqsubseteq S^\dagger,$$

and

$$\begin{array}{llll} \forall S_2 \in \mathcal{C}(\Sigma_2) & S_1^\circ \wedge S_2 \sqsubseteq S^\dagger & \Rightarrow & \mathrm{L}(S_1^\circ \wedge S_2/G) \subseteq \mathrm{L}(S_1^\circ \wedge S_2^\circ/G), \text{ and} \\ \forall S_1 \in \mathcal{C}(\Sigma_1) & S_1 \wedge S_2^\circ \sqsubseteq S^\dagger & \Rightarrow & \mathrm{L}(S_1 \wedge S_2^\circ/G) \subseteq \mathrm{L}(S_1^\circ \wedge S_2^\circ/G). \end{array}$$

In game theory, controllers have conflicting optimization criteria, whereas in team theory all controllers try to optimize the same cost criterion. The notion of Nash equilibrium has been introduced in game theory. In team theory it is also known as a person by person optimal solution.

In team theory, under certain convexity conditions, a set of controllers is maximal if and only if it is a Nash equilibrium [49]. This equivalence is quite useful because it is relatively easier to determine a Nash equilibrium than a maximum.

The following example shows that for discrete event systems the Nash equilibrium condition is not sufficient to guarantee maximality.

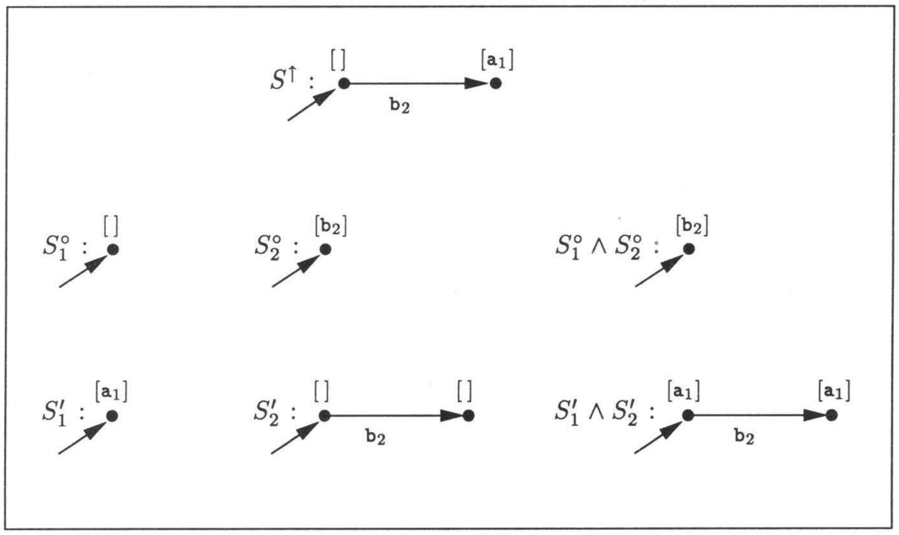


Figure 6.3: The pair (S_1°, S_2°) is a Nash equilibrium, yet it is not maximal.

Example 6.20 Consider the supremal supervisor $S^\uparrow$ and the decentralized implementation (S_1°, S_2°) given in Figure 6.3. $\Sigma_1 = \{\mathtt{a}_1\}$, $\Sigma_2 = \{\mathtt{b}_2\}$. All events are controllable. It is not difficult to check that the pair (S_1°, S_2°) is a Nash equilibrium. However, it is not maximal, because the pair (S_1', S_2') is a solution with a strictly larger controlled language.

For discrete event systems we need the stronger condition of strong Nash equilibrium to guarantee maximality of a pair of supervisors.

Definition 6.21 A pair of supervisors $(S_1^\circ, S_2^\circ) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is called a *strong Nash equilibrium* if it is a Nash equilibrium, and for all $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$

$$\mathrm{L}(S_1 \wedge S_2/G) = \mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \quad \Rightarrow \quad (S_1, S_2) \text{ is a Nash equilibrium.}$$

By Proposition 6.7 $\mathrm{L}(S_1 \wedge S_2/G) = \mathrm{L}(S_1^\circ \wedge S_2^\circ/G)$ and $S_1^\circ \wedge S_2^\circ \sqsubseteq S^\uparrow$ together imply that $S_1 \wedge S_2 \sqsubseteq S^\uparrow$.

Theorem 6.22 *A pair of supervisors* $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ *is maximal if and only if it is a strong Nash equilibrium.*

Proof. (Strong Nash $\Rightarrow$ Maximal.) Assume $(S_1^\circ, S_2^\circ) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is strong Nash but not maximal. Then there exists a pair $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ such that $S_1 \wedge S_2 \sqsubseteq S^\uparrow$ and $\mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \subsetneqq \mathrm{L}(S_1 \wedge S_2/G)$. Define $S_1^\square \in \mathcal{C}(\Sigma_1)$ by

$$\gamma(S_1^\square, s_1) \quad = \quad \gamma(S_1^\circ, s_1) \cup \gamma(S_1, s_1), \quad \text{for all } s_1 \in \mathrm{p}_1(\mathrm{L}(G)).$$

We will prove the following points.

1. $\mathrm{L}(S_1^\square/G) = \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G)$,
2. $S_1^\square \wedge S_2^\circ \sqsubseteq S^\uparrow$,
3. $S_1^\square \wedge S_2 \sqsubseteq S^\uparrow$,
4. $\mathrm{L}(S_1^\circ \wedge S_2^\circ/G) = \mathrm{L}(S_1^\square \wedge S_2^\circ/G)$,
5. $\mathrm{L}(S_1^\square \wedge S_2^\circ/G) \subseteq \mathrm{L}(S_1^\square \wedge S_2/G)$.

(Point 1.) This point will be proven by complete induction. The initial step follows from $\varepsilon \in \mathrm{L}(S_1^\square/G)$ and $\varepsilon \in \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G)$. For the inductive step let $s \in \mathrm{L}(S_1^\square/G)$ and $s \in \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G)$. Then

$$\begin{aligned}
& s\sigma \in \mathrm{L}(S_1^\square/G) \\
\iff \quad & s\sigma \in \mathrm{L}(G),\ \sigma \notin \gamma(S_1^\square, \mathrm{p}_1(s)) \\
\iff \quad & s\sigma \in \mathrm{L}(G),\ \sigma \notin \gamma(S_1^\circ, \mathrm{p}_1(s)),\ \sigma \notin \gamma(S_1, \mathrm{p}_1(s)) \\
\iff \quad & s\sigma \in \mathrm{L}(S_1^\circ/G),\ s\sigma \in \mathrm{L}(S_1/G) \\
\iff \quad & s\sigma \in \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G).
\end{aligned}$$

It follows that $\mathrm{L}(S_1^\square/G) = \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G)$.

(Points 2 and 4.) From Point 1 and Proposition 6.10 it follows that

$$\begin{aligned}
\mathrm{L}(S_1^\square \wedge S_2^\circ/G) &= \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2^\circ/G) \\
&= \mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \cap \mathrm{L}(S_1/G) \\
&= \qquad [\mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \subseteq \mathrm{L}(S_1 \wedge S_2/G) \text{ and} \\
&\qquad \text{By Proposition 6.10 } \mathrm{L}(S_1 \wedge S_2/G) \subseteq \mathrm{L}(S_1/G)] \\
&\quad\ \mathrm{L}(S_1^\circ \wedge S_2^\circ/G).
\end{aligned}$$

This proves point 4. Point 2 follows from $S_1^\circ \wedge S_2^\circ \sqsubseteq S^\uparrow$ and Proposition 6.7.

(Point 3.) From Point 1 and Proposition 6.10 it follows that

$$\begin{aligned}
\mathrm{L}(S_1^\square \wedge S_2/G) &= \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G) \\
&\subseteq \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G) \\
&= \mathrm{L}(S_1 \wedge S_2/G) \\
&\subseteq \mathrm{L}(S^\uparrow/G).
\end{aligned}$$

So, by Proposition 6.7, $S_1^\square \wedge S_2 \sqsubseteq S^\uparrow$.

(Point 5.) From Point 4 it follows that

$$\begin{aligned}
\mathrm{L}(S_1^\square \wedge S_2^\circ/G) &= \mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \\
&= \qquad\qquad [\mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \subseteq \mathrm{L}(S_1 \wedge S_2/G)] \\
&\quad\ \mathrm{L}(S_1^\circ \wedge S_2^\circ/G) \cap \mathrm{L}(S_1 \wedge S_2/G) \\
&= \mathrm{L}(S_1^\circ/G) \cap \mathrm{L}(S_2^\circ/G) \cap \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G) \\
&\subseteq \mathrm{L}(S_1^\square/G) \cap \mathrm{L}(S_2/G) \\
&= \mathrm{L}(S_1^\square \wedge S_2/G).
\end{aligned}$$

As the pair (S_1°, S_2°) is strong Nash, it follows from point 4 that $(S_1^\square, S_2^\circ)$ is Nash. So, by point 3 and the definition of Nash, $\mathrm{L}(S_1^\square \wedge S_2/G) \subseteq \mathrm{L}(S_1^\square \wedge S_2^\circ/G)$. Then, from points 4 and 5 $\mathrm{L}(S_1^\circ \wedge S_2^\circ/G) = \mathrm{L}(S_1^\square \wedge S_2^\circ/G) = \mathrm{L}(S_1^\square \wedge S_2/G)$. As (S_1°, S_2°) is strong Nash, the pair $(S_1^\square, S_2)$ is Nash. So

$$\mathrm{L}(S_1 \wedge S_2/G) \quad\subseteq\quad \mathrm{L}(S_1^\square \wedge S_2/G) \quad=\quad \mathrm{L}(S_1^\circ \wedge S_2^\circ/G).$$

But this contradicts our assumption that $\mathrm{L}(S_1^\circ \wedge S_2^\circ) \subsetneq \mathrm{L}(S_1 \wedge S_2/G)$. We can conclude that if (S_1°, S_2°) is strong Nash then it is maximal.

(Maximal $\Rightarrow$ Strong Nash.) Assume $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ is maximal but not strong Nash. Then there exists a pair $(S_1', S_2') \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ such that $\mathrm{L}(S_1' \wedge S_2'/G) = \mathrm{L}(S_1 \wedge S_2/G)$ and (S_1', S_2') is not Nash. So

$$\exists S_1'' \in \mathcal{C}(\Sigma_1) \quad \text{s.t.} \quad S_1'' \wedge S_2' \sqsubseteq S^\uparrow \quad \text{and} \quad \mathrm{L}(S_1'' \wedge S_2'/G) \not\subseteq \mathrm{L}(S_1' \wedge S_2'/G)$$

or

$$\exists S_2'' \in \mathcal{C}(\Sigma_2) \quad \text{s.t.} \quad S_1' \wedge S_2'' \sqsubseteq S^\uparrow \quad \text{and} \quad \mathrm{L}(S_1' \wedge S_2''/G) \not\subseteq \mathrm{L}(S_1' \wedge S_2'/G).$$

Assume, without loss of generality, that such an S_2'' exists. Let $S_2^\square \in \mathcal{C}(\Sigma_2)$ be defined by

$$\gamma(S_2^\square, s_2) \;=\; \begin{cases} \gamma(S_2', s_2) \cap \gamma(S_2'', s_2), & \text{if } s_2 \in \mathrm{p}_2(\mathrm{L}(S_2'/G)) \text{ and } s_2 \in \mathrm{p}_2(\mathrm{L}(S_2''/G)), \\ \gamma(S_2', s_2), & \text{if } s_2 \in \mathrm{p}_2(\mathrm{L}(S_2'/G)) \text{ and } s_2 \notin \mathrm{p}_2(\mathrm{L}(S_2''/G)), \\ \gamma(S_2'', s_2), & \text{if } s_2 \notin \mathrm{p}_2(\mathrm{L}(S_2'/G)) \text{ and } s_2 \in \mathrm{p}_2(\mathrm{L}(S_2''/G)), \\ \Sigma_{2,c}, & \text{otherwise.} \end{cases}$$

We will prove the following points.

1. $\mathrm{L}(S_2^\square/G) = \mathrm{L}(S_2'/G) \cup \mathrm{L}(S_2''/G)$,
2. $S_1' \wedge S_2^\square \sqsubseteq S^\uparrow$,
3. $\mathrm{L}(S_1' \wedge S_2'/G) \subseteq \mathrm{L}(S_1' \wedge S_2^\square/G)$,

4. $L(S_1' \wedge S_2''/G) \subseteq L(S_1' \wedge S_2^{\square}/G)$.

(Point 1.) This point will be proven by complete induction. The initial step follows from $\varepsilon \in L(S_2^{\square}/G)$ and $\varepsilon \in L(S_2'/G) \cup L(S_2''/G)$. For the inductive step let $s \in L(S_2^{\square}/G)$ and $s \in L(S_2'/G) \cup L(S_2''/G)$. Trace s can be in one of the three sets $L(S_2'/G) \cap L(S_2''/G)$, $L(S_2'/G) - L(S_2''/G)$, or $L(S_2''/G) - L(S_2'/G)$. If $s \in L(S_2'/G) \cap L(S_2''/G)$, then

$$\begin{aligned}
& s\sigma \in L(S_2^{\square}/G) \\
\Longleftrightarrow \quad & s\sigma \in L(G) \wedge \sigma \notin \gamma(S_2^{\square}, p_2(s)) \\
\Longleftrightarrow \quad & s\sigma \in L(G) \ \wedge \ \big(\sigma \notin \gamma(S_2', p_2(s)) \vee \sigma \notin \gamma(S_2'', p_2(s))\big) \\
\Longleftrightarrow \quad & s\sigma \in L(S_2'/G) \vee s\sigma \in L(S_2''/G) \\
\Longleftrightarrow \quad & s\sigma \in L(S_2'/G) \cup L(S_2''/G).
\end{aligned}$$

If $s \in L(S_2'/G)$ but $s \notin L(S_2''/G)$, then

$$\begin{aligned}
& s\sigma \in L(S_2^{\square}/G) \\
\Longleftrightarrow \quad & s\sigma \in L(G) \wedge \sigma \notin \gamma(S_2^{\square}, p_2(s)) \\
\Longleftrightarrow \quad & s\sigma \in L(G) \wedge \sigma \notin \gamma(S_2', p_2(s)) \\
\Longleftrightarrow \quad & s\sigma \in L(S_2'/G) \\
\Longleftrightarrow \quad & s\sigma \in L(S_2'/G) \cup L(S_2''/G).
\end{aligned}$$

A similar reasoning holds if $s \in L(S_2''/G)$ but $s \notin L(S_2'/G)$. Hence, it follows that $L(S_2^{\square}/G) = L(S_2'/G) \cup L(S_2''/G)$.

(Points 2, 3, and 4) From point 1 and Proposition 6.10 it follows that

$$\begin{aligned}
L(S_1' \wedge S_2^{\square}/G) &= L(S_1'/G) \cap \big(L(S_2'/G) \cup L(S_2''/G)\big) \\
&= \big(L(S_1'/G) \cap L(S_2'/G)\big) \cup \big(L(S_1'/G) \cap L(S_2''/G)\big) \\
&= L(S_1' \wedge S_2'/G) \cup L(S_1' \wedge S_2''/G).
\end{aligned}$$

This directly proves points 3 and 4. Point 2 follows from $S_1' \wedge S_2' \sqsubseteq S^{\uparrow}$, $S_1' \wedge S_2'' \sqsubseteq S^{\uparrow}$ and Proposition 6.7.

As (S_1, S_2) is maximal, so is (S_1', S_2'). Then, by point 3, $L(S_1' \wedge S_2'/G) = L(S_1' \wedge S_2^{\square}/G)$. From point 4 it follows that $L(S_1' \wedge S_2''/G) \subseteq L(S_1' \wedge S_2'/G)$. But this contradicts our assumption that $L(S_1' \wedge S_2''/G) \not\subseteq L(S_1' \wedge S_2'/G)$. Hence it can be concluded that if (S_1, S_2) is maximal then it is strong Nash. $\square$

Consider two pairs of supervisors to be *control equivalent* if their controlled languages are equal. Then a pair of supervisors is maximal if and only if all control equivalent pairs are Nash equilibria. Let the *control equivalence class* corresponding with the language $K \subseteq L(G)$ be the set of pairs for which the controlled language is equal to K. A prefix closed and decomposable language can be considered maximal if and only if all pairs in its corresponding control equivalence class are Nash equilibria.

If the event sets Σ_1 and Σ_2 are disjoint, then a weaker condition can be found to characterize maximal solutions. Define $(\widehat{S}_1, \widehat{S}_2)$ as the pair of most restrictive supervisors in the control equivalence class of (S_1, S_2).

Definition 6.23 Let $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$. The supervisor $\widehat{S}_1 \in \mathcal{C}(\Sigma_1)$ is defined by

$$\gamma(\widehat{S}_1, s_1) \quad = \quad \{\sigma \in \Sigma_c(S_1) : s_1\sigma \notin \mathrm{p}_1(\mathrm{L}(S_1 \wedge S_2/G)\}, \quad \forall s_1 \in \mathrm{p}_1(\mathrm{L}(G)).$$

The supervisor $\widehat{S}_2 \in \mathcal{C}(\Sigma_2)$ is defined equivalently.

Supervisor $\widehat{S}_1$ can be seen as the most restrictive supervisor of all supervisors S_1' for which there exists a supervisor S_2' such that $\mathrm{L}(S_1' \wedge S_2'/G) = \mathrm{L}(S_1 \wedge S_2/G)$. That is, if such an S_1' disables event σ after trace s then $s\sigma$ is not an element of $\mathrm{L}(S_1'/G) \supseteq \mathrm{L}(S_1' \wedge S_2'/G) = \mathrm{L}(S_1 \wedge S_2/G)$. So $\widehat{S}_1$ will also disable this event.

First it needs to be proven that $(\widehat{S}_1, \widehat{S}_2)$ is a solution and that it is control equivalent with (S_1, S_2).

Proposition 6.24 *Let $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ and let $(\widehat{S}_1, \widehat{S}_2)$ be defined as above. Then*

1. $\widehat{S}_1 \wedge \widehat{S}_2 \sqsubseteq S^\uparrow$, *and*
2. $\mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G) = \mathrm{L}(S_1 \wedge S_2/G)$.

Proof. (Point 2, $\mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G) \subseteq \mathrm{L}(S_1 \wedge S_2/G)$.) First we will prove by induction that $\mathrm{L}(\widehat{S}_1/G) \subseteq \mathrm{L}(S_1/G)$. The initial step follows from $\varepsilon \in \mathrm{L}(\widehat{S}_1/G)$ and $\varepsilon \in \mathrm{L}(S_1/G)$. For the inductive step let $s \in \mathrm{L}(\widehat{S}_1/G)$ and $s \in \mathrm{L}(S_1/G)$.

$$\begin{aligned}
& s\sigma \in \mathrm{L}(\widehat{S}_1/G) \\
\Rightarrow \quad & s\sigma \in \mathrm{L}(G) \wedge \sigma \notin \gamma(\widehat{S}_1, \mathrm{p}_1(s)) \\
\Rightarrow \quad & s\sigma \in \mathrm{L}(G) \wedge \left(\sigma \notin \Sigma_c(S_1) \vee \mathrm{p}_1(s)\sigma \in \mathrm{p}_1(\mathrm{L}(S_1 \wedge S_2/G))\right) \\
\Rightarrow \quad & \hfill [\mathrm{L}(S_1 \wedge S_2/G) \subseteq \mathrm{L}(S_1/G)] \\
& s\sigma \in \mathrm{L}(G) \wedge \left(\sigma \notin \Sigma_c(S_1) \vee \mathrm{p}_1(s)\sigma \in \mathrm{p}_1(\mathrm{L}(S_1/G))\right) \\
\Rightarrow \quad & s\sigma \in \mathrm{L}(G) \wedge \left(\sigma \notin \Sigma_c(S_1) \vee \sigma \notin \gamma(S_1, \mathrm{p}_1(s))\right) \\
\Rightarrow \quad & \hfill [\sigma \notin \Sigma_c(S_1) \Rightarrow \sigma \notin \gamma(S_1, \mathrm{p}_1(s))] \\
& s\sigma \in \mathrm{L}(G) \wedge \sigma \notin \gamma(S_1, \mathrm{p}_1(s)) \\
\Rightarrow \quad & s\sigma \in \mathrm{L}(S_1/G).
\end{aligned}$$

By symmetry it follows that $\mathrm{L}(\widehat{S}_2/G) \subseteq \mathrm{L}(S_2/G)$. So

$$\begin{aligned}
\mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G) &= \mathrm{L}(\widehat{S}_1/G) \cap \mathrm{L}(\widehat{S}_2/G) \\
&\subseteq \mathrm{L}(S_1/G) \cap \mathrm{L}(S_2/G) \\
&= \mathrm{L}(S_1 \wedge S_2/G).
\end{aligned}$$

(Point 2, $L(S_1 \wedge S_2/G) \subseteq L(\widehat{S}_1 \wedge \widehat{S}_2/G)$.) First it will be proven by induction that $L(S_1 \wedge S_2/G) \subseteq L(\widehat{S}_1/G)$. The initial step follows from $\varepsilon \in L(S_1 \wedge S_2/G)$ and $\varepsilon \in L(\widehat{S}_1/G)$. For the inductive step let $s \in L(S_1 \wedge S_2/G)$ and $s \in L(\widehat{S}_1/G)$.

$$\begin{aligned}
& s\sigma \in L(S_1 \wedge S_2/G) \\
\Rightarrow \quad & \sigma \notin \Sigma_1 \vee \left(\sigma \in \Sigma_1 \wedge p_1(s\sigma) = p_1(s)\sigma \in p_1(L(S_1 \wedge S_2/G))\right) \\
\Rightarrow \quad & \qquad\qquad\qquad\qquad [\text{Construction of } \gamma(\widehat{S}_1, p_1(s))] \\
& \sigma \notin \Sigma_1 \vee \left(\sigma \in \Sigma_1 \wedge \sigma \notin \gamma(\widehat{S}_1, p_1(s))\right) \\
\Rightarrow \quad & \qquad\qquad\qquad\qquad [\gamma(\widehat{S}_1, p_1(s)) \subseteq \Sigma_1] \\
& \sigma \notin \gamma(\widehat{S}_1, p_1(s)) \\
\Rightarrow \quad & \qquad\qquad\qquad\qquad [s \in L(\widehat{S}_1/G) \text{ and } s\sigma \in L(G)] \\
& s\sigma \in L(\widehat{S}_1/G).
\end{aligned}$$

By symmetry it follows that $L(S_1 \wedge S_2/G) \subseteq L(\widehat{S}_2/G)$. So

$$\begin{aligned}
L(S_1 \wedge S_2/G) & \subseteq L(\widehat{S}_1/G) \cap L(\widehat{S}_2/G) \\
& = L(\widehat{S}_1 \wedge \widehat{S}_2/G).
\end{aligned}$$

(Point 1.) This follows directly from Point 2 and Proposition 6.7. □

Theorem 6.25 *Let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Let $(S_1, S_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$. Let $(\widehat{S}_1, \widehat{S}_2) \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ be defined by Definition 6.23. Then (S_1, S_2) is maximal if and only if $(\widehat{S}_1, \widehat{S}_2)$ is a Nash equilibrium.*

Proof. (Maximal $\Rightarrow$ Nash.) If (S_1, S_2) is maximal then by Theorem 6.22 (S_1, S_2) is a strong Nash equilibrium, which by points 1 and 2 of Proposition 6.24 implies that $(\widehat{S}_1, \widehat{S}_2)$ is a Nash equilibrium.

(Nash $\Rightarrow$ Maximal.) Assume $(\widehat{S}_1, \widehat{S}_2)$ is a Nash equilibrium, but (S_1, S_2) is not maximal. Then, by point 2 of Proposition 6.24 $(\widehat{S}_1, \widehat{S}_2)$ is not maximal. There exists a pair $(S_1', S_2') \in \mathcal{C}(\Sigma_1) \times \mathcal{C}(\Sigma_2)$ such that $S_1' \wedge S_2' \sqsubseteq S^{\uparrow}$ and $L(\widehat{S}_1 \wedge \widehat{S}_2/G) \subsetneqq L(S_1' \wedge S_2'/G)$. We will first prove that

$$\widehat{S}_1 \wedge S_2' \sqsubseteq S^{\uparrow} \quad \text{and} \quad S_1' \wedge \widehat{S}_2 \sqsubseteq S^{\uparrow}.$$

It will be proven by induction that $L(\widehat{S}_1/G) \subseteq L(S_1'/G)$. The initial step follows from $\varepsilon \in L(\widehat{S}_1/G)$ and $\varepsilon \in L(S_1'/G)$. For the inductive step let $s \in L(\widehat{S}_1/G)$ and $s \in L(S_1'/G)$.

$$\begin{array}{ll}
\multicolumn{2}{l}{s\sigma \in \mathrm{L}(\widehat{S}_1/G)} \\
\Rightarrow & s\sigma \in \mathrm{L}(G) \ \wedge \ \sigma \notin \gamma(\widehat{S}_1, \mathrm{p}_1(s)) \\
\Rightarrow & s\sigma \in \mathrm{L}(G) \ \wedge \ \big(\sigma \notin \Sigma_c(S_1) \vee \mathrm{p}_1(s)\sigma \in \mathrm{p}_1(\mathrm{L}(S_1 \wedge S_2/G))\big) \\
\Rightarrow & \qquad\qquad\qquad [\mathrm{L}(S_1 \wedge S_2/G) \subseteq \mathrm{L}(S_1' \wedge S_2'/G) \subseteq \mathrm{L}(S_1'/G)] \\
 & s\sigma \in \mathrm{L}(G) \ \wedge \ \big(\sigma \notin \Sigma_c(S_1) \vee \mathrm{p}_1(s)\sigma \in \mathrm{p}_1(\mathrm{L}(S_1'/G))\big) \\
\Rightarrow & s\sigma \in \mathrm{L}(G) \ \wedge \ \big(\sigma \notin \Sigma_c(S_1) \vee \sigma \notin \gamma(S_1', \mathrm{p}_1(s))\big) \\
\Rightarrow & \qquad\qquad\qquad [\sigma \notin \Sigma_c(S_1) \ \Rightarrow \ \sigma \notin \gamma(S_1', \mathrm{p}_1(s))] \\
 & s\sigma \in \mathrm{L}(G) \ \wedge \ \sigma \notin \gamma(S_1', \mathrm{p}_1(s)) \\
\Rightarrow & s\sigma \in \mathrm{L}(S_1'/G).
\end{array}$$

It follows that $\mathrm{L}(\widehat{S}_1/G) \subseteq \mathrm{L}(S_1'/G)$. Now

$$\begin{aligned}
\mathrm{L}(\widehat{S}_1 \wedge S_2'/G) &= \mathrm{L}(\widehat{S}_1/G) \cap \mathrm{L}(S_2'/G) \\
&\subseteq \mathrm{L}(S_1'/G) \cap \mathrm{L}(S_2'/G) \\
&= \mathrm{L}(S_1' \wedge S_2'/G) \\
&\subseteq \mathrm{L}(S^{\uparrow}/G).
\end{aligned}$$

So, by Proposition 6.7, $\widehat{S}_1 \wedge S_2' \sqsubseteq S^{\uparrow}$. It follows by symmetry that $S_1' \wedge \widehat{S}_2 \sqsubseteq S^{\uparrow}$.

As $\mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G) \subsetneqq \mathrm{L}(S_1' \wedge S_2'/G)$ there exist a trace $s \in \mathrm{L}(S_1' \wedge S_2'/G)$ such that $s \notin \mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G)$. Let $v\sigma$ be the prefix of s such that $\sigma \in \Sigma$, $v \in \mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G)$, and $v\sigma \notin \mathrm{L}(\widehat{S}_1 \wedge \widehat{S}_2/G)$. Assume without loss of generality that $\sigma \in \Sigma_2$. Then, by the assumption that $\Sigma_1 \cap \Sigma_2 = \emptyset$, $\sigma \notin \Sigma_1$. So $\sigma \notin \gamma(\widehat{S}_1, \mathrm{p}_1(v)) \subseteq \Sigma_1$. Thus $v\sigma \in \mathrm{L}(\widehat{S}_1/G)$. As $v\sigma \in \mathrm{L}(S_1' \wedge S_2'/G) \subseteq \mathrm{L}(S_2'/G)$, it follows that $v\sigma \in \mathrm{L}(\widehat{S}_1 \wedge S_2'/G)$. But this contradicts the fact that $(\widehat{S}_1, \widehat{S}_2)$ is a Nash equilibrium. Hence we can conclude that if $(\widehat{S}_1, \widehat{S}_2)$ is a Nash equilibrium, then (S_1, S_2) is maximal. □

A prefix closed and decomposable language $K \subseteq \mathrm{L}(G)$ can be considered maximal if and only if the pair of most restricting supervisors in the control equivalence class corresponding with language K is a Nash equilibrium.

6.6. Construction of Nash Equilibria

Theorems 6.22 and 6.25 give characterizations of the maximal solutions in terms of Nash equilibria. However, they do not state how Nash equilibria can be obtained. For dynamic games in the field of game and team theory, a necessary condition for a Nash equilibrium can be given by the coupled Bellman-Hamilton-Jacobi equations. A solution to these equations is under certain additional conditions also sufficient for a Nash equilibrium. An algorithm for the construction of a solution is known [19]. It alternately keeps one of the controllers fixed and tries to optimize the other. At each iteration only one of the controllers is optimized. For dynamic games it is not guaranteed that the algorithm converges. And if it converges, it is not guaranteed that it does so in a finite number of steps.

For supervisory control the Bellman-Hamilton-Jacobi equations are not applicable. Yet, the algorithm can still be used. At each iteration one of the supervisors is kept fixed and the other is optimized. Only one supervisor is synthesized in each step. This can be seen as a supervisory control problem for a single supervisor. The combination of the fixed supervisor and the uncontrolled system is taken as the uncontrolled system for this control problem. As only one supervisor is synthesized (and all controllable events are observable) an unique optimal solution exists. In the next iteration this optimal supervisor is taken fixed and the other supervisor is optimized. This procedure is repeated until the pair of supervisors remains invariant.

Assume without loss of generality that S_1 is the supervisor which is kept fixed. The optimal solution is characterized by the supremal normal and controllable sublanguage of $\mathrm{L}(S_1/G)$. Let K be this language. The supremal supervisor S_2 with respect to the uncontrolled system S_1/G is defined by

$$\gamma(S_2, s_2) \quad = \quad \{\sigma \in \Sigma_c(S_2) : s_2\sigma \notin \mathrm{p}_2(K)\}, \quad \forall s_2 \in \mathrm{p}_2(\mathrm{L}(G)). \qquad (6.1)$$

If S_2 is kept fixed then S_1 is computed equivalently.

Lemma 6.26 *Let S_1 be the supervisor which is kept fixed. Let S_2 and K be as defined above. Then $\mathrm{L}(S_1 \wedge S_2/G) = K$.*

Proof. The proof will be by complete induction. As $\varepsilon \in \mathrm{L}(S_1 \wedge S_2/G)$ and $\varepsilon \in K$ the initial step is satisfied. For the inductive step let $s \in \mathrm{L}(S_1 \wedge S_2/G)$ and $s \in K$.

$$\begin{aligned}
& s\sigma \in \mathrm{L}(S_1 \wedge S_2/G) \\
\iff\quad & s\sigma \in \mathrm{L}(S_1/G) \wedge \sigma \notin \gamma(S_2, \mathrm{p}_2(s)) \\
\iff\quad & s\sigma \in \mathrm{L}(S_1/G) \;\wedge\; \big(\sigma \notin \Sigma_c(\Sigma_2) \vee \mathrm{p}_2(s)\sigma \in \mathrm{p}_2(K)\big) \\
\iff\quad & \hfill \big[s \in K \text{ and } K \text{ is controllable}\big] \\
& s\sigma \in \mathrm{L}(S_1/G) \;\wedge\; \big(s\sigma \in K \vee \mathrm{p}_2(s)\sigma \in \mathrm{p}_2(K)\big) \\
\iff\quad & \hfill [K \subseteq \mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K))] \\
& s\sigma \in \mathrm{L}(S_1/G) \wedge s\sigma \in \mathrm{p}_2^{-\uparrow}(\mathrm{p}_2(K)) \\
\iff\quad & \hfill [K \text{ is normal w.r.t. } \mathrm{L}(S_1/G)] \\
& s\sigma \in K.
\end{aligned}$$
□

The algorithm is described by the following four steps.

1. Choose a pair of most restrictive supervisors (S_1^0, S_2^0) as starting point of the algorithm. Take for instance the pair of most restrictive supervisors corresponding with the fully decentralized solution. Let $j = 0$.

2. If j is even then let S_2^{j+1} be the supremal supervisor with respect to uncontrolled system S_1^j/G and event set Σ_2. Let $S_1^{j+1} = S_1^j$. If j is odd then let S_1^{j+1} be the supremal supervisor with respect to uncontrolled system S_2^j/G and event set Σ_1. Let $S_2^{j+1} = S_2^j$.

3. If $(S_1^{j+1}, S_2^{j+1}) \neq (S_1^j, S_2^j)$ then increment j and continue with step 2.

First it will be shown that all pairs of supervisors (S_1^j, S_2^j) are most restricting.

Lemma 6.27 *Let (S_1^j, S_2^j) be most restrictive. Then (S_1^{j+1}, S_2^{j+1}) obtained in the second step of the algorithm is also most restrictive.*

Proof. Assume without loss of generality that j is odd. So $S_2^{j+1} = S_2^j$ and S_1^{j+1} is the supremal supervisor with respect to S_2^j/G. Let $K^j = \mathrm{L}(S_1^j \wedge S_2^j/G)$ and $K^{j+1} = \mathrm{L}(S_1^{j+1} \wedge S_2^{j+1}/G)$. Comparing (6.1) with Definition 6.23 it is not difficult to see that S_1^{j+1} is most restrictive with respect to K^{j+1}. Supervisor $S_2^{j+1} = S_2^j$ is most restrictive with respect to language K^j. It remains to show that it is most restrictive with respect to K^{j+1}.

$$\begin{aligned}
&\sigma \in \gamma(S_2^{j+1}, s_2) = \gamma(S_2^j, s_2) \\
&\quad\Rightarrow \quad \sigma \in \Sigma_c(S_2^j) \wedge s_2\sigma \notin \mathrm{L}(S_2^j/G) \\
&\quad\Rightarrow \qquad\qquad\qquad\qquad [K^{j+1} \subseteq \mathrm{L}(S_2^j/G)] \\
&\qquad\quad \sigma \in \Sigma_c(S_2^j) \wedge s_2\sigma \notin K^{j+1}.
\end{aligned}$$

As K^{j+1} is supremal it follows that $K^j \subseteq K^{j+1}$.

$$\begin{aligned}
&\sigma \notin \gamma(S_2^{j+1}, s_2) = \gamma(S_2^j, s_2) \\
&\quad\Rightarrow \qquad\qquad\qquad\qquad [S_2^j \text{ is most restrictive w.r.t. } K^j] \\
&\qquad\quad \sigma \notin \Sigma_c(S_2^j) \vee s_2\sigma \in K^j \\
&\quad\Rightarrow \qquad\qquad\qquad\qquad [K^j \subseteq K^{j+1}] \\
&\qquad\quad \sigma \notin \Sigma_c(S_2^j \vee s_2\sigma \in K^{j+1}.
\end{aligned}$$

It follows that S_2^{j+1} is most restrictive with respect to K^{j+1}. And thus (S_1^{j+1}, S_2^{j+1}) is most restrictive. □

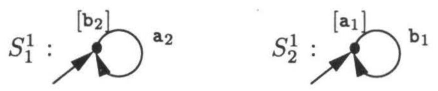
Next it will be shown that if $(S_1^{j+1}, S_2^{j+1}) = (S_1^j, S_2^j)$ then (S_1^j, S_2^j) forms a Nash equilibrium. So if Σ_1 and Σ_2 are disjoint, then this pair is a maximal solution.

Theorem 6.28 *Let $j \in \mathbb{N}$ and let $S_1^j, S_2^j, S_1^{j+1}, S_2^{j+1}$ be constructed by the algorithm above. If $(S_1^{j+1}, S_2^{j+1}) = (S_1^j, S_2^j)$ then (S_1^j, S_2^j) forms a Nash equilibrium.*

Proof. Assume without loss of generality that j is odd. Then, according to the second step of the algorithm $S_2^{j+1} = S_2^j$ and S_1^{j+1} is the supremal supervisor with respect to S_2^j/G. As $S_1^{j+1} = S_1^j$ it follows that S_1^j is optimal if S_2^j is kept fixed. This proves the first part of the Nash equilibrium condition.

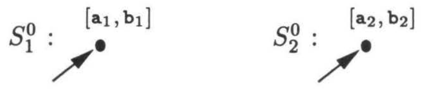
From the previous iteration of the algorithm it follows that $S_1^j = S_1^{j-1}$ and that S_2^j is the supremal supervisor with respect to S_1^{j-1}/G. In the next iteration supervisor S_2^{j+2} will be synthesized. Supervisor S_2^{j+2} is the optimal solution with respect to $S_1^{j+1}/G = S_1^j/G = S_1^{j-1}/G$. So S_2^{j+2} will be equal to S_2^j. Supervisor S_2^j is optimal if S_1^j is kept fixed. This proves the second part of the Nash equilibrium condition. And thus (S_1^j, S_2^j) is a Nash equilibrium. □

Example 6.29 Consider the system described in Example 6.18 and Figure 6.2. Take the pair of most restrictive supervisors corresponding with the fully decentralized solution as starting point of the algorithm. In this case $S_1^0 = \text{proj}(S^\uparrow, \Sigma_1)$ and $S_2^0 = \text{proj}(S^\uparrow, \Sigma_2)$. Let $\Sigma_1 = \{\mathtt{a}_1, \mathtt{b}_1\}$, $\Sigma_2 = \{\mathtt{a}_2, \mathtt{b}_2\}$, and $\Sigma_c = \{\mathtt{a}_1, \mathtt{a}_2\}$. Note that the event sets Σ_1 and Σ_2 are disjoint. First S_1^0 is kept fixed and the optimal supervisor S_2^1 with respect to the uncontrolled system S_1^0/G is derived.

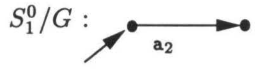


Next, S_2^1 is kept fixed and the optimal supervisor S_1^2 with respect to the uncontrolled system S_2^1/G is derived. In subsequent steps the pair of supervisors remains invariant. The pair (S_1^1, S_2^1) is a Nash-equilibrium, and therefore, according to Theorem 6.25, a maximal solution.

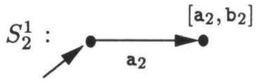
Now, consider a slight alteration of the control problem. Let $\Sigma_c = \Sigma$ and let the rest be unchanged. Take, as before, the pair of most restrictive supervisors corresponding with the fully decentralized solution as starting point. In this case also the b-events are disabled.



First S_1^0 is kept fixed.

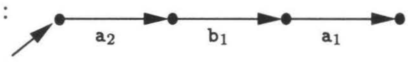


Supervisor S_2^1 is the optimal supervisor with respect to S_1^0/G.



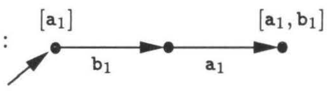
Next, this supervisor is kept fixed.

S_2^1/G :



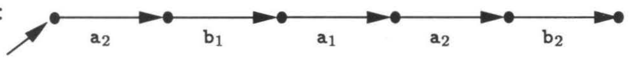
The optimal supervisor S_1^2 with respect to S_2^1/G is computed.

S_1^2 :



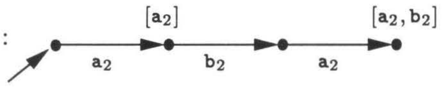
In the next step this supervisor is kept fixed.

S_1^2/G :



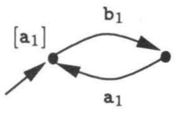
Supervisor S_2^3 is the optimal supervisor with respect to S_1^2/G.

S_2^3 :

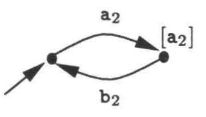


The algorithm will converge to the limit pair (S_1^*, S_2^*).

S_1^* :



S_2^* :



However, this solution will not be obtained in a finite number of steps.

The example shows that a small change in the parameters of the problem may lead to a different solution. It may even cause the algorithm to become non-halting.

Up till now these particularities are not fully understood. Further research is needed to adapt the algorithm such that it always converges in a finite number of steps. Also further research is required to understand the relationship between the initial parameters and the eventual solution. It would be ideal if the algorithm could synthesize a finite representation of all maximal solutions. However, up till now it is not even certain whether such a finite representation exists.

Chapter 7
Concluding Remarks

In this thesis several aspects of discrete event control problems are discussed. The direction of the research has been motivated by control and design problems for layered network architectures. The results, however, are generic in nature. They hold for a much more general class of discrete event control problems.

In the first part of this thesis issues concerning the layered structure are treated. The control problem is to find a supervisor such that the controlled system can replace the specification. In particular it is required that the controlled system cannot deadlock in situations in which the specification cannot deadlock either.

In Chapter 2 a control framework is introduced which is capable of dealing with discrete event control problems for layered architectures. It guarantees deadlock freeness of the controlled system in any environment, it allows for powerful nondeterministic specifications that can even be partial, it forms a sound basis for modular control, it can handle nondeterministic systems, and it can handle partial observations.

The key concept in this framework is nondeterminism. Literally nondeterminism means: not fully determined. Nondeterminism represents uncertainty. A main concept in this first part of the thesis is that a supervisory control framework for layered architectures should be able to handle the uncertainty caused by unknown environments, the uncertainty caused by partial specifications, the uncertainty in the underlying uncontrolled system, and the uncertainty caused by partial observations. All these uncertainties are related to nondeterminism.

In this thesis we have chosen failure semantics as basis for our framework. However, we believe that this choice is not crucial. Any semantics that can guarantee the correct behavior of nondeterministic systems in any environment can be used. In [2] a number of semantics are discussed that satisfy these requirements. We are confident that the results in this thesis can be extended easily to those semantics.

Another conclusion that can be drawn from the first chapters is that nondeterminism is nothing to be afraid of. When dealing with uncertainty there is no reason not to use nondeterminism. The results from Chapter 2 till 5 show that nondeterminism can be treated elegantly in a flexible and powerful framework. It is very much a question whether issues such as divergence caused by projection can be treated in a deterministic setting.

In Chapter 3 a general supervisor synthesis method is proposed. It generates a nondeterministic supervisor which represents all solutions. This supervisor is called the supremal supervisor. All solutions to the control problem can be derived easily from this supremal supervisor. It is shown that the deterministic least restrictive solution is the deterministic supervisor defined by the language of the supremal supervisor.

The complexity of the synthesis algorithm is linear in sizes of the behavior state spaces of G and E. Of course, systems will usually not be defined by a behavior state representation. Further research is necessary to study the complexity of transformations from, for instance, a representation based on process algebras to a behavior state representation. The transformation from an automaton model to a behavior state representation is known to be worst case exponential. However, for practical systems this transformation might not be a problem.

The complexity of automatic synthesis is comparable to that of automatic verification. Moreover, automatic synthesis and automatic verification can be applied to the same kind of design problems. In these design problems the correctness of an implementation must be guaranteed. Automatic synthesis and automatic verification are competitors. We believe that automatic synthesis has a lot to offer in this competition. With automatic synthesis all the information available from the uncontrolled system and the specification is taken into account. This information is automatically processed into the supremal supervisor. From this supremal other solutions can be derived. The key issue is that *all* available information is taken into account. With automatic verification a designer first has to derive a solution from scratch. Afterwards this solution is checked. These two steps are independent. Information used for the design is not used for the verification and vice versa. With automatic synthesis these steps are integrated. Automatic synthesis can be seen as the ultimate form of verification based design methods.

The automatic synthesis procedure has much resemblance with automatic verification procedures. It is therefore expected that complexity reducing methods developed for automatic verification can also be applied to automatic synthesis. More research is necessary in this direction.

In Chapter 4 the supervisory control problem with partial specification is discussed. It is shown how it can be reduced to the basic supervisory control problem with full specification. Special care has to be taken for divergent traces. A condition, denoted bounded recurrence, is introduced to handle these traces. Bounded recurrence limits the number of times that a trace can recur (come back) to the same state. It is shown that the external behavior of the controlled system is not restricted by the bounded recurrence condition.

Note that the necessity of the bounded recurrence condition is not caused by the use of failure semantics. Bounded recurrence is needed to properly handle the nondeterministic phenomena caused by partial specifications.

In Chapter 5 it is shown that the supervisory control problem with partial observation for which all controllable events are observable has a straightforward solution. It shows the strength of the synthesis method presented in Chapter 3 that this control problem can be easily handled.

In the rest of the chapter it is shown that all discrete event control problems can be modeled such that all controllable events are observable. The key issue is that the control law of the supremal supervisor will depend on the previous control actions. The system has to be modeled such that these control actions appear in the traces observed by the supervisor. It is shown that this is always possible.

In Chapter 6 the decentralized supervisory control problem problem is discussed. It is shown that all maximal solutions are characterized by the strong Nash condition. If the event sets of the supervisors are disjunct then a pair of supervisors is maximal if an equivalent canonical solution forms a Nash equilibrium. The results illustrate conceptual properties that may help to construct practical synthesis methods for decentralized control problems.

The key difficulty with the synthesis of decentralized controllers is that the control law of one supervisor needs to be known to synthesize the other supervisor. This leads to a cyclic reasoning that is hard to break. The difficulty when synthesizing one controller is that the control law of the other supervisor is still uncertain. As stated before, uncertainty can be modeled by nondeterminism. Maybe that the use of nondeterministic supervisors to model the uncertainty during the design phase can help to derive practical synthesis methods.

Control problems for layered network architectures can be decomposed into two dimensions. Control problems in the vertical dimension, corresponding with control problems for layered architecture, can be solved elegantly using the control framework presented in this thesis. Control problems in the horizontal dimension, corresponding with decentralized control problems, are still unsolvable. However, we have presented fundamental results that may be useful to solve decentralized control problems.

Bibliography

1. F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity, An Algebra For Discrete Event Systems.* Wiley, 1992.
2. J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.
3. W. de Bakker, J. P. de Roever, W. and G. Rozenberg, editors. *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness* Mook, The Netherlands, *1989*, LNCS 430. Springer Verlag, 1990.
4. S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, 1993.
5. J. Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36:844–851, 1989.
6. T. Başar and G. J. Olsder. *Dynamic Noncooperative Game Theory.* Academic Press, New York, 1982.
7. G. Birkhoff. *Lattice Theory.* A.M.S., Providence, 1967.
8. B. A. Brandin and W. M. Wonham. The supervisory control of timed discrete event systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, 1994.
9. V. van Breusegem, L. Ben-Naoum, and R. K. Boel. Maximally permissive feedback controls for controlled state machines. Report X, CESAME, Université Catholique de Louvain, Louvain-la-Neuve, 1992.

10. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

11. S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, LNCS 197, pages 281–305, 1985.

12. International Conference on Computer aided Verification. The proceedings to date have been published as *Lecture Notes in Computer Science (LNCS)* 407 (1989), 531 (1990), 575 (1991), 663 (1992), 697 (1993), 818 (1994), 939 (1995).

13. R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, 1988.

14. G. Cohen and J.-P. Quadrat, editors. *11th International Conference on Analysis and Optimization of Systems, Discrete Event Systems*, Sophia-Antipolis, 1994, Lecture Notes in Control and Information Sciences 199. Springer, 1994.

15. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

16. M. D. DiBenedetto, A. Saldanha, and A. Sangiovanni-Vincentelli. Model matching for finite state machines. In *Proceedings of the 33th Conference on Decision and Control*, Orlando, Florida, pages 3117–3124, 1994.

17. M. D. DiBenedetto, A. Saldanha, and A. Sangiovanni-Vincentelli. Strong model matching for finite state machines. In A. Isidori, e.a., editors, *Proceedings of the 3rd European Control Conference* Rome, Italy, pages 2027–2034, 1995.

18. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

19. Olsder G. J. Comments on a numerical procedure for the solution of differential games. *IEEE Transactions on Automatic Control*, 20:704–705, 1975.

20. N. B. Hadj-Alouane, S. Lafortune, and F. Lin. Variable lookahead supervisory control with state information. *IEEE Transactions on Automatic Control*, 39(12):2398–2410, 1994.

21. E. Haghverdi and K. Inan. Verification by consecutive projections. In M. Diaz and R. Groz, editors, *FORTE '92, proceedings of the IFIP*, Perros-Guirec, pages 465–478, 1993.

22. Y. C. Ho. Team decision theory and information structures. *Proceedings of the IEEE*, 68:644–654, 1980.

23. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

24. G. J. Hoffmann and H. Wong-Toi. The input-output control of real-time discrete event systems. Report ISL/GFF/92-1, Information Systems Laboratory, Stanford, 1992.

25. L. E. Holloway and B. H. Krogh. Controlled petri nets: A tutorial survey. In Cohen and Quadrat [14].

26. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

27. K. Inan. Nondeterministic supervision under partial observation. In Cohen and Quadrat [14], pages 39–48.

28. ISO: *Information Processing Systems — Open Systems Interconnection* — LOTOS — *A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, ISO 8807, International Organisation for Standardisation, Geneva, CH, 1989.

29. B. Jonsson and K. G. Larsen. On the complexity of equation solving. In *Proceedings of TAPSOFT*, LNCS 493, 1991.

30. P. Kozák and W. M. Wonham. Fully decentralized solutions of supervisory control problems. Systems Control Group Report No. 9310, University of Toronto, Dept. of Electrical Engineering, 1993.

31. R. Kumar, V. Garg, and S. I. Marcus. On controllability and normality of discrete event dynamical systems. *Systems & Control Letters*, 17(3):157–168, 1991.

32. R. Kumar and M. A. Shayman. Avoiding blocking in prioritized synchronization based control of nondeterministic systems. In Cohen and Quadrat [14], pages 49–58.

33. R. P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

34. S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control*, 33(5):439–447, 1988.

35. K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS)*, Philadelphia, USA, pages 108–117, 1990.

36. F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

37. F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions on Automatic Control*, 35(12):1330–1337, 1990.

38. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

39. R. Milner. *A Calculus of Communicating Systems.* LNCS 92. Springer, 1980.

40. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

41. J. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

42. A. Overkamp. Supervisory control for nondeterministic systems. In Cohen and Quadrat [14], pages 59–65.

43. A. Overkamp. Control of nondeterministic discrete event systems using failure semantics. In A. Isidori e.a., editors, *Proceedings of the 3rd European Control Conference* Rome, Italy, pages 2778–2783, 1995.

44. A. Overkamp. Supervisory control using failure semantics and partial specifications. Accepted for publication in *Transactions on Automatic Control*, 1996.

45. A. Overkamp and J. H. van Schuppen. Control of discrete event systems - research at the interface of control theory and computer science. In K. Apt, L. Schrijver, and N. Temme, editors, *From Universal Morphisms to Megabytes: A Baayen Space Odyssey*, pages 453–467, Stichting Mathematisch Centrum, Amsterdam, December 1994.

46. A. Overkamp and J. H. van Schuppen. A characterization of maximal solutions for decentralized discrete event control problems. In *Proceedings of the 3rd Workshop on Discrete Event Systems (WODES 96)*, Edinburgh, UK, August 19–21, 1996.

47. J. L. Peterson. *Petri Net Theory and Modeling of Systems.* Prentice-Hall, 1981.

48. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

49. R. Radner. Team decision problems. *Annals of mathematical statistics*, 33:857–881, 1962.

50. P. J. Ramadge and W. M. Wonham. Supervision of discrete event processes. In *Proceedings of the 21th Conference on Decision and Control*, Orlando, Florida, pages 1228–1229, 1982.

51. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25:206–230, 1987.

52. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

53. K. Rudie. *Decentralized Control of Discrete-Event Systems.* PhD thesis, Department of Electrical Engineering, University of Toronto, 1992. Also

available as System Control Group Report No. 9209.

54. K. Rudie and W. M. Wonham. Protocol verification using discrete-event systems. In *Proceedings of the 31st Conference on Decision and Control*, Tucson, Arizona, pages 3770–3777, New York, 1992. IEEE Press.

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

56. M. A. Shayman and R. Kumar. Supervisory control of nondeterministic systems with driven events via prioritized synchronization and trajectory models. *SIAM Journal on Control and Optimization*, 33(2):469–497, 1995.

57. A. Tanenbaum. *Computer Networks, second edition*. Prentice Hall, 1988.

58. J. G. Thistle. *Control of Infinite Behaviour of Discrete-Event Systems*. PhD thesis, University of Toronto, 1991. Available as Systems Control Group Report No. 9012.

59. J. G. Thistle. Logical aspects of control of discrete-event systems: A survey of tools and techniques. In Cohen and Quadrat [14], pages 3–15.

60. K. C. Wong and W. M. Wonham. Hierarchical and modular control of discrete-event systems. In *Proceedings Thirtieth Annual Allerton Conference on Communication, Control and Computing*, University of Illinois, pages 604–613, 1992.

61. K. C. Wong and W. M. Wonham. Hierarchical control of timed discrete-event systems. In *Proceedings of the 2nd European Control Conference* Groningen, The Netherlands, pages 509–512, 1993.

62. W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

63. W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

64. H. Zhong. *Hierarchical Control of Discrete-Event Systems*. PhD thesis, Department of Electrical Engineering, University of Toronto, 1992.

65. H. Zhong and W. M. Wonham. On consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35:1125–1134, 1990.

66. H. Zimmermann. OSI reference model — the ISO model of architecture for open systems interconnection. *IEEE Transactions on Computers*, 28:425–432, 1980.

Samenvatting

De titel van dit proefschrift luidt in het Nederlands: ' Discrete-gebeurtenissen-regeling gemotiveerd door gelaagde netwerkarchitecturen'. Discrete-gebeurtenissen-regeling betreft het bepalen van regelaars voor systemen met discrete gebeurtenissen. Voorbeelden van discrete gebeurtenissen in dit soort systemen zijn: iemand drukt op het knopje van de lift op de derde verdieping, een bericht komt aan in een knooppunt van een telecommunicatienetwerk, en een machine in een flexibele productiecel levert een gereed product op. Voor dit soort systemen is de tradionele systeem- een regeltheorie niet geschikt. Die theorie houdt zich bezig met continu variabele systemen. Dat zijn systemen waarin veranderingen veel gelijkmatiger optreden.

Sinds het begin van de jaren tachtig wordt er onderzoek verricht naar regelingen voor systemen met discrete gebeurtenissen. In navolging van de grondleggers P. G. Ramadge en W. M. Wonham zijn allerlei regelproblemen geformuleerd en opgelost. Het doel in al deze regelproblemen is een regelaar te vinden zodanig dat een vooraf gegeven systeem in combinatie met deze regelaar voldoet aan een vooraf gegeven specificatie. Het vakgebied staat tegenwoordig bekend onder de term: supervisie-theorie.

Veel systemen die in de informatica gebruikt worden kunnen beschouwd worden als systemen met discrete gebeurtenissen. Het uitgangspunt van het onderzoek dat ten grondslag aan dit proefschrift ligt, is uit te zoeken hoe de resultaten van supervisie-theorie toe te passen zijn op ontwerpproblemen uit de informatica. In het bijzonder wordt gekeken naar ontwerpproblemen voor computer- en communicatienetwerken. De resultaten zijn ook toe te passen op andere ontwerpproblemen voor systemen met discrete gebeurtenissen.

Traditioneel wordt binnen de supervisie-theorie alleen naar determinis-

tische systemen gekeken. Bij dit soort systemen kan uit de sequentie van opeenvolgende gebeurtenissen bepaald worden in welke toestand het systeem zich bevindt. Hieruit volgt weer welke sequenties van gebeurtenissen het systeem hierna kan uitvoeren. Binnen de informatica wordt voornamelijk met niet-deterministische systemen gewerkt. Dit soort systemen kan vanuit één toestand naar meerdere mogelijke toestanden overgaan, waarbij iedere overgang overeenkomt met eenzelfde waargenomen gebeurtenis. Niet-determinisme biedt een algemene manier om onzekerheid over het gedrag van systemen weer te geven. Als eerste stap is het nodig om de resultaten van supervisie-theorie uit te breiden naar niet-deterministische systemen.

Een sequentie van gebeurtenissen wordt een spoor genoemd. Deterministische systemen worden adequaat beschreven door de verzameling van sporen die het systemen kan genereren. Deze verzameling wordt de taal van een systeem genoemd. De meeste resultaten van supervisie-theorie voor deterministische systemen zijn geformuleerd in termen van talen. Voor een beschrijving van het gedrag van niet-deterministische systemen is de taal alleen niet voldoende. De taal beschrijft welke gebeurtenissen na een spoor mogelijk op kunnen treden. Het is noodzakelijk om ook te beschrijven welke gebeurtenissen mogelijk niet kunnen gebeuren. Deze uitgangspunten liggen ten grondslag aan de beschrijvingsmethodiek 'failure semantics'. Failure semantics is geintroduceerd door C. A. R. Hoare als een wiskundige onderbouwing van de programmeertaal CSP. In dit proefschrift wordt failure semantics gebruikt om het gedrag van niet-deterministische systemen met discrete gebeurtenissen te beschrijven.

Gelaagde Netwerk Architecturen

Computer- en communicatienetwerken zijn zeer complexe systemen. Toch zijn informatici in staat gebleken om dit soort systemen te ontwerpen. De belangrijkste stap om complexe systemen hanteerbaar te maken is ze in kleinere onderdelen op te delen. Ieder onderdeel apart is eenvoudiger te ontwerpen dan het geheel. Het is noodzakelijk om ervoor te zorgen dat naderhand alle onderdelen weer goed samenwerken.

Netwerken zijn opgedeeld in een gelaagde structuur. Iedere laag verzorgt een specifiek sub-doel met betrekking tot het oversturen van berichten. Dit sub-doel wordt de service van deze laag genoemd. Zo zorgt een bepaalde laag ervoor dat er geen berichten verloren gaan. Een andere laag zorgt ervoor dat berichten op de juiste bestemming aankomen. Iedere laag levert zijn service door berichten tussen de verschillende knooppunten heen en weer te sturen. Dit berichtenverkeer wordt het protocol genoemd. Voor het versturen van de berichten wordt de service van de onderliggende laag gebruikt. De laag die ervoor zorgt dat er geen berichten verloren gaan verstuurt zijn berichten via de onderliggende laag. Deze onderliggende laag kan niet garanderen dat een bericht ook daadwerkelijk aankomt. Door berichten opnieuw te sturen en door gebruik van extra 'regel'-berichten kan de bovenliggende laag toch garanderen dat ieder bericht uiteindelijk aankomt.

Het protocol-ontwerp-probleem vertoont veel overeenkomst met het regelprobleem voor systemen met discrete gebeurtenissen. Beschouw de service die geleverd moet worden als de specificatie, het protocol als de regelaar die ontworpen moet worden, en de service van de onderliggende laag als ongeregeld systeem. Dan kan het protocol-ontwerp-probleem beschouwd worden als een discrete gebeurtenissen-regelprobleem.

De gelaagde structuur zoals hierboven beschreven, stelt een aantal eisen aan het regelprobleem die tot nu toe nog niet zijn beschouwd binnen het vakgebied van de supervisie-theorie. Een laag moet namelijk in staat zijn samen te werken met de bovenliggende lagen. De service die geboden wordt door laag n zal door laag $n+1$ gebruikt worden om de service voor laag $n+1$ aan te kunnen bieden. Bij het ontwerp van het protocol voor laag $n+1$ is er van uit gegaan dat dit protocol de service van laag n kan gebruiken. Het protocol voor laag n moet de service van laag n zodanig implementeren dat laag $n+1$ deze kan gebruiken. Het protocol voor laag $n+1$ zal in het algemeen op een andere plaats ontworpen worden dan het protocol voor laag n. Tijdens het ontwerp van het protocol van laag n is het protocol van laag $n+1$ niet bekend. De service van laag n moet zodanig geïmplementeerd worden dat ieder systeem dat goed met de service samenwerkt ook goed werkt met de implementatie ervan.

Beschouw de volgende implementatie relatie. Systeem A reduceert systeem B als de taal van systeem A bevat is in de taal van systeem B, en A zich meer deterministisch gedraagt dan B. We zeggen dat A B implementeert als A een reductie is van B. Het basis regelprobleem kan nu als volgt worden gedefinieerd. Gegeven een specificatie E en een ongeregeld systeem G, zoek een regelaar S, zodanig dat de combinatie van G en S systeem E reduceert. Met deze probleemdefinitie zal een protocol alleen dan een oplossing zijn, als het in combinatie met de onderliggende laag goed samenwerkt met de bovenliggende laag.

De oplossingsmethode voor bovenstaand regelprobleem is vergelijkbaar met bestaande oplossingsmethoden voor deterministische systemen. De methode genereerd de zogenaamde supremale regelaar. Deze regelaar is het supremum van de verzameling regelaars die een oplossing vormen voor het regelprobleem. Een regelaar is een oplossing van het regelprobleem dan en slechts dan als het een reductie is van de supremale regelaar. De complexiteit van de methode is lineair in de grootte van de toestandsruimten van het ongeregelde systeem en de specificatie.

Partiële Specificaties

Specificaties behoren onafhankelijk te zijn van de implementatie. Dat wil zeggen: gebeurtenissen die niet relevant zijn voor de omgeving van het systeem worden niet in de specificatie gebruikt. Op deze manier zal de specificatie onafhankelijk zijn van de gebruikte implementatie. De specificatie kan voor verschillende implementaties (bijvoorbeeld van verschillende fabrikanten) gebruikt worden. Aangezien niet alle gebeurtenissen die in het ongeregelde systeem voorkomen beschreven worden in de specificatie, wordt de specificatie partieel genoemd. Het bijbehorende regelprobleem kan vertaald worden naar het basis regelprobleem dat hiervoor beschreven is. Hierbij moet echter rekening gehouden worden met onbegrensde sequenties van interne gebeurtenissen. Dit zijn gebeurtenissen die niet in de specificatie gebruikt worden. Voor de omgeving van het systeem doen deze onbegrensde sequenties zich voor als een weigering van het systeem om een externe gebeurtenis uit te voeren. Dit wordt een 'deadlock' genoemd. Het protocol moet ervoor zorgen dat in voorkomende gevallen alleen een begrensd aantal implementatie gebeurtenissen op kunnen treden. Hiertoe is het begrip 'begrensde recursie' ingevoerd. Begrensd recursieve protocollen begrenzen het aantal interne gebeurtenissen wanneer dit noodzakelijk is. Door de oplossingsruimte te beperken tot begrensd recursieve regelaars kan het regelprobleem met partiële specificatie vertaald worden naar een regelprobleem met volledige specificatie. Een begrensd recursieve regelaar is een oplossing van het regelprobleem met partiële specificatie dan en slechts dan als het een oplossing is van het bijbehorende regelprobleem met volledige specificatie. De restrictie tot begrensd recursieve regelaars is niet van invloed op het externe gedrag van de mogelijke regelaars.

Partiële Observaties

In sommige gevallen kan een regelaar niet alle gebeurtenissen van het ongeregelde systeem waarnemen. Zo kan er een fout optreden die van belang is voor het gedrag van het systeem, maar die niet waargenomen kan worden door de regelaar.

Een gebeurtenis wordt regelbaar genoemd als de regelaar de mogelijkheid heeft deze gebeurtenis te blokkeren. De gebeurtenis kan dan niet optreden. Resultaten in de literatuur laten zien dat het regelprobleem veel moeilijker is als niet alle regelbare gebeurtenissen waarneembaar zijn.

Partiële observatie is sterk gerelateerd aan niet-determinisme. Enerzijds kan niet-determinisme gezien worden als het gevolg van het niet volledig observeerbaar zijn van alle gebeurtenissen in het systeem. Anderzijds kan niet-determinisme gebruikt worden om de onzekerheid veroorzaakt door partiële observatie te modelleren. Het raamwerk zoals dat in dit proefschrift is gepresenteerd is in staat met de niet-deterministische effecten veroorzaakt door partiële observatie om te gaan. Indien all regelbare gebeurtenissen waarneembaar zijn, is het bepalen van een oplossing voor het regelprobleem met partiële observatie een niet al te moeilijke uitbreiding van de oplossingsmethode voor het regelprobleem met volledige observatie.

Indien er regelbare gebeurtenissen zijn die niet waarneembaar zijn, dan kan het probleem zodanig gehermodelleerd worden dat alle regelbare gebeurtenissen waarneembaar zijn. Hiertoe worden de acties die nodig zijn om regelbare niet-waarneembare gebeurtenissen te regelen, gemodelleerd als regelbare en waarneembare gebeurtenissen. Alle discrete-gebeurtenissen-regelproblemen kunnen zodanig gemodelleerd worden dat alle regelbare gebeurtenissen waarneembaar zijn.

Gedecentraliseerde regeling

De gelaagde structuur van netwerken komt overeen met één dimensie van het protocol-ontwerp-probleem. De andere dimensie wordt gevormd door het decentrale karakter van netwerken. Een protocol wordt niet geïmplementeerd als één monolitisch geheel, maar in stukken verdeeld over de knooppunten van het netwerk. Dit aspect van het protocol-ontwerp-probleem komt overeen met het gedecentraliseerde regelprobleem. Doel van dit regelprobleem is het vinden van een verzameling van regelaars zodanig dat iedere regelaar maar een deel van het systeem kan observeren en beïnvloeden. De combinatie van alle regelaars tezamen met het ongeregelde systeem moet de specificatie implementeren.

Dit regelprobleem is zeer moeilijk. Alleen beperkte resultaten zijn bekend. Het regelprobleem is gerelateerd aan regelproblemen uit het vakgebied van de spel- en teamtheorie. In deze gebieden is het begrip 'Nash equilibrium' geïntroduceerd. Een oplossing is een Nash-equilibrium als iedere regelaar afzonderlijk zich niet kan verbeteren wanneer de overige regelaars onveranderd blijven. Een oplossing wordt een sterk Nash-equilibrium genoemd als iedere verzameling regelaars die een equivalent geregeld systeem opleverd, ook een Nash-equilibrium is. Een oplossing is een maximale oplossing van het decentrale discrete-gebeurtenissen-regelprobleem dan en slechts dan als het een sterk Nash-equilibrium is.

Indien de gebeurtenissen die de regelaars kunnen observeren disjunct zijn, dan kan de conditie verzwakt worden. In dit geval is een oplossing maximaal dan en slecht dan als een canonieke equivalente oplossing een Nash-equilibrium is.