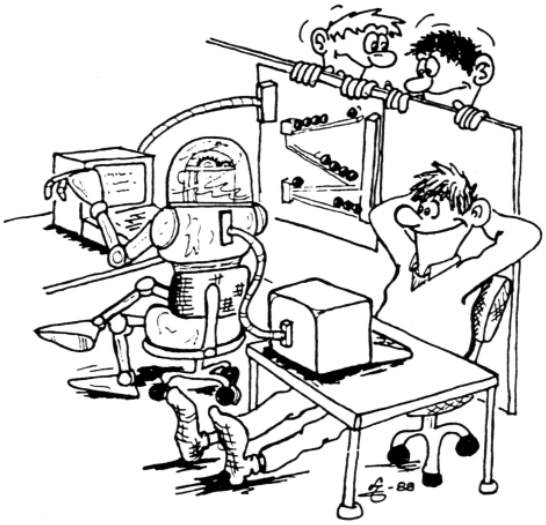
# On Object Oriented Non-Deterministic Supervisory Control

Martin Fabian



Ph. D. Thesis
Technical Report 282, December 1995

Control and Automation Laboratory
Department of Signals and Systems
Chalmers University of Technology
Sweden

*In memory of*
*Terész and János Fábián*

# Abstract

Implementation of complex discrete event fabrication processes can be considerably simplified by use of general reusable software modules representing the physical components. At the same time, construction of the control system can be facilitated by use of formal methods for automatic generation of the control laws. These two aspects can be joined into a general concept with object-oriented modeling and control law synthesis as foundations. The goal is to allow an operator to specify operation lists describing the required sequences of operations for the manufacturing of the product, independently of constraints given by a specific plant. With a suitable model of the capabilities and constraints of the resources of that plant, a product route can be automatically generated from the operation list. Such a product route describes all available paths through the system, for each type of product, irrespective of any other type of product that may be simultaneously present within the production system. Given a set of product routes and a model of the plant, control laws guaranteeing production according to those product specifications can be synthesized.

Based on the supervisory control theory, using interleaved product routes as specification, we show how such control laws can be synthesized. An added complexity is that the a specification becomes non-deterministic, in the sense that the same string of events can lead to different system states. We show that the supervisory control theory can be used with non-deterministic specifications assuming certain properties. We also describe an object-oriented modeling approach to discrete event fabrication processes. It is shown that the properties that have been defined as necessary for the non-deterministic supervisory approach are immediate by the modeling approach. Thus, we show that the approach to non-deterministic supervisory control can be combined with object-oriented modeling techniques, and so we have a powerful framework for implementing control of large and complex discrete event fabrication processes.

**Keywords.** Supervisory Control, Object-Oriented Modeling, Flexible Manufacturing, Finite State Automata

# Published Papers

The work presented in this thesis is based on the following technical reports, papers and conference presentations.

1. Fabian, M., B. Lennartson, *Control of Manufacturing Systems; An Object Oriented Approach*, 7th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'92, Toronto, Canada, May 1992.

2. Fabian, M., B. Lennartson, *Object Oriented Structuring of Real Time Control Systems*, 31st IEEE Conference on Decision and Control, CDC'92, Tuscon, Arizona, December 1992.

3. Fabian, M., *Reusable Software Components for Structuring Real-Time Control Systems*, Technical Report 143L, Licentiate Thesis, Control Engineering laboratory, Chalmers University of Technology, Göteborg, Sweden, January 1993.

4. Fabian, M., B. Lennartson, *Distributed Objects for Real Time Control Systems*, 12th IFAC World Congress, Sydney, Australia, July 1993.

5. Fabian, M., B. Lennartson, *Petri Nets and Control Synthesis; An Object Oriented Approach*, 3rd International Intelligent Manufacturing Systems Symposium, IMS'94, Vienna, Austria, June 1994.

6. Fabian, M., B. Lennartson, *Object-Oriented Supervisory Control with a Class of Nondeterministic Specifications*, 33rd IEEE Conference on Decision and Control, CDC'94, Buena Vista, Florida, December 1994.

7. Fabian, M., B. Lennartson, *A Class of Nondeterministic Marked Specification for Supervisory Control*, 3rd SIAM Conference on Control and Its Applications, St Louis, Missouri, USA, April 1995.

8. Adlemo, A., S-A. Andréasson, M. Fabian, P. Gullander, B. Lennartson, *Towards a Truly Flexible Manufacturing System*, Computer Engineering Practice, Vol 3, No 4, 545-554, April 1995.

9. Tittus, M., M. Fabian, *Automated Generation of Plant Specific Recipes in Batch Control*, ICCI'95, Hong Kong, June 1995

10. Fabian, M., B. Lennartson, *A Class of Nondeterministic Specification for Supervisory Control*, 3rd IFAC European Control Conference, ECC'95, Rome, Italy, September 1995.

11. Fabian, M., B. Lennartson, *Petri Net Constructs for High Level Operation Lists*, 8th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'95, Beijing, China, October 1995.

12. Fabian, M., P. Gullander, B. Lennartson, S-A Andréasson, A. Adlemo, *Dynamic Products in Control of an FMS Cell*, 8th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'95, Beijing, China, October 1995.

13. Fabian, M., B. Lennartson, *Applying Supervisory Control Theory to Discrete Event Systems Modeled by Object Oriented Principles*, INRIA/IEEE Conference on Emerging Technologies and Factory Automation, ETFA'95, Paris, France, October 1995.

14. Tittus, M., M. Fabian, B. Lennartson, *Controlling and Coordinating Recipes in Batch Applications*, 34th IEEE Conference on Decision and Control, CDC'95, New Orleans, Louisiana, USA, December 1995.

15. Fabian, M., B. Lennartson, *Applying Supervisory Control Theory to Discrete Event Systems Modeled by Object Oriented Priniples; How and When*, Internal Report CTH/RT/I-95/008, Control Engineering Laboratory, Chalmers University of Technology, Göteborg, Sweden, 1995.

These will be referenced as, for instance, [14] in the following text

# Acknowledgements

No man is an island.

This work would not have been possible without the support, guidance, inspiring influence, motivation, help, etc., of many other individuals. The first to mention is, of course, my mentor and supervisor Dr. Bengt Lennartson. I am grateful to him for his time invested in me and this work, and for believing in me, even at times when I myself did not. He also pushed me on when it was necessary, and often came up with the right phrase to trigger my thoughts in the right direction; I think, sometimes without even understanding my incoherent ramblings at all. His help in editing this thesis I consider invaluable. Thank you, Bengt. (Chapter 5 was a tough one, wasn't it?)

Dr. Michael Tittus (yes, you're a *Doctor* now, Michael) has been a great counterpart in many discussions. He also helped out in the editing process, and generally made lots of fun jokes (well, *some* of them at least). Thanks, Michael

Professor Bo Egardt pointed out several errors in the propositions and proofs. He asked the innocent question that revealed a serious error. Suddenly it seemed I would not even make it due to this. Thanks (!?). In the end it all worked out, and I am thankful for the all the support. (No tennis game, though, please.)

Dr. Christos Cassandras, of University of Massachusetts at Amhurst, took the time to read this thesis and to explore some of its inner regions, to oppose me so I could defend myself for the dissertation. Thank you. Hopefully, I did get all the pieces together of this puzzle I am laying, and hopefully I put them in their right places with their right sides up.

Professor Bo Egardt, Dr. Bengt Lennartson, Dr. Michael Tittus and me, we are all at the Control Engineering Department. Of course, we are not alone there. All you people at the department have, in one way or another, contributed to this work, though you may not know it. Gunnar Berg, Claes Breitholtz, Catharina Forssén, Adam Lagerberg, Annika Leonard, Torbjörn Liljenvall, Claës Lindeborg, Olof Lindgärde, Ulla-Britt Nilsson, Stefan Petterson, Bengt Schmidtbauer, Lars Wallin, Torsten Wik.

The following individuals, in no particular order, deserve a special "thank you", for various reasons briefly mentioned.

Lars Jansson for keeping the computers going and the dog happy, and for letting me use the car when that uncontrollable event occurred. Lemmy, for inducing that extra strength when needed. Thommy Ekelund, for helping me get rid of that strength. Per Gullander, for all the help with the GeRMs (hope they haven't spread). Domo arigato, gozaimasu. Bonzo the Dog, for just being so happy all the time (sometimes *too* happy, Lars!). Kopparbergs Elk, for the inspiration.

Lars Jansson also made the illustration of what supervisory control can turn out to be, if we do not watch out.

Last, but certainly not least, my life companion Katarina Johnsen. She understood that I had to work so much, without understanding with what. So many things had to be held over, I will make it up to you now, love.

If I forgot someone, thank you too.

And, Tony–you know what you shouldn't do.

August 10, 2000

# Contents

# Chapter 1

# Introduction

As pointed out by Fox (1992), modern corporations in the industrialized world must seek new paths to be able to compete in the coming decade. These paths ought to result in safe and efficient ways to manufacture customized, high-quality, environmentally benign and technically advanced products. This calls for increasingly flexible manufacturing processes. However, industrial demands for a fast project-implement-run cycle, often limits both the generality and the flexibility of the designed systems. Due to the complexity of the problem, the resulting systems often become not only expensive but also inflexible, despite the fact that their components are themselves highly flexible.

The main reason for this is that the control system is heavily influenced by the product routes relevant at the time of implementation. Implementing only the required routes and no more, gives (in some sense) the minimum amount of work. This minimal work load is often at the cost, though, of decreased flexibility and reusability. For new product routes and for implementation of new systems, much (if not all) code has to be re-implemented. Thus, this approach would (in some other sense) really require a maximum amount of work. For instance, robots and machine tools can be used flexibly in a large variety of applications, while the control software that synchronizes these components most often cannot function with machine tools or products other than those it was designed for, see Sargent (1993). In order to lower costs and increase flexibility, we need a truly flexible manufacturing system that can be reused without the need to reprogram the controller whenever a change in production is introduced.

Flexibility can be incorporated on different time-scales. A long time-scale flexibility permits incorporation of new or different equipment within the system, without having to extensively reprogram the control system. Flexibility on a medium time-scale allows incorporation of new products within the manufacturing system, and flexibility on a very short time-scale permits rescheduling of the manufacturing system on-line. When implementing control of production systems, some high-level support for modeling is of a great advantage. This high-level support should provide means to build a system appropriate for all three levels of flexibility, described above.

*Object-oriented modeling* has shown to be a valuable tool for structuring complex systems and easing their implementation by providing general software components reusable in different applications with little or no alteration, see for instance Joannis (1992) and Jobling (1994). The systems resulting from object-oriented modeling consist of independent communicating modules. Thus, object-oriented modeling supports the incorporation

of new equipment, without extensive reprogramming of already functional modules. In this way, the long time-scale aspect of flexibility is supported.

Flexibility on the medium time-scale is supported by *distributed product specification*. Each product is an autonomous entity coexisting with other products simultaneously present within the system. However, we can certainly do without the added complexity of having to specify independent product routes with regard to other products that may be existing at the time of production. It is favorable to be able to specify a distinct product route irrespective of any other products, and have some underlying system tie it all together.

Finally, the *automatic synthesis of control laws* supports the very short time-scale aspect of flexibility. For the sake of reliability and safety of the constructed system, this synthesis should be supported by a rigid foundation of formal methods. Given an object-oriented model of the system and a number of independent product specifications, together with other specifications on the systems behavior, safe and correct control laws must be synthesized. Since a manufacturing system can be regarded as a discrete event process (DEP), this foundation of formal methods can be found in the supervisory control theory (SCT) initiated by Ramadge (1987) and Wonham (1987).

Being a relatively new discipline within control theory, DEPs have attracted much attention both in the systems modeling area and within the control-law synthesis field. Shlaer (1992) and Adiga (1993) describe methodologies of modeling DEPs, based on object-oriented approaches. A dynamic system is modeled as being composed of objects, whose dynamic behavior is expressed by state automata. These objects communicate by means of events representing state transitions. Ramadge (1987) and Wonham (1987) have, with the SCT, provided a unifying framework for synthesis of control laws for DEPs. Kumar (1991) and Balemi (1992), among others, have proposed their own variations of the SCT, based on different interpretations of the interaction between the controlling and the controlled processes. Giua (1991) bases an approach on Petri nets. Even so, until now there has been very little (possibly no) work done in merging the two domains of object-oriented modeling and supervisory control synthesis.

In this work we will show how object-oriented modeling of flexible manufacturing and assembly systems can be used with the SCT. The object-oriented modeling approach builds on the ability to identify and extract the general behavior of production resources into reusable software models, one for each different class of manufacturing device. These reusable models offer generalized functions on a high abstraction level, using lower level, specific instruction sequences to implement that behavior. In this way the synchronizing aspects of the required control will be separated from the control of the actual devices. The general, high-level functionality is represented by message driven DEPs. The messages are modeled as *events*. Thus, the *plant* to be controlled consists of a number of independent (but possibly coupled) DEPs, operating concurrently.

The specification of the product routes will likewise be given as DEPs. Each type of product can be specified without regard for any other product simultaneously present within the system, even though they may require use of the same processing equipment. The product routes thus specified encompass a subset of the events offered by the plant, not necessarily disjoint between different types of products. The concurrently executing product routes can thus be seen as a specification for the plant to exhibit a certain behavior, and it is up to some underlying control system, the *supervisor*, to guarantee

that this behavior is actually accomplished.

To synthesize a supervisor with a global view of the controlled system, the individual product route specifications have to be composed into a joint specification on the overall system. However, the products are to run asynchronously with respect to each other, though synchronously with respect to the plant. The concurrent sharing of the machining resources by the products can be modeled by *interleaving*, see Hoare (1985). The joint specification is the interleaving of the individual product route specifications, which leads to a *non-deterministic* specification; non-deterministic in the sense that a choice between several products simultaneously wanting to share a resource may arise. Only one of these will be allowed access to the shared resource, and the choice between them is non-deterministic. Non-determinism in this sense means that one and the same sequence of events may lead to any of a number of states. It will in this work be shown that the SCT is valid for the certain class of non-deterministic specification arising in systems as described above. Furthermore, a supervisor does exist and is algorithmically constructable.

The resulting system contains three basic types of objects all described as DEPs; *internal resources* that are models of the actual machining equipment; *product individuals* that model the physical workpieces; and a *controller* that controls the behavior of the system, so as to fulfill the specification of having the products satisfactorily produced. The controller operates within the boundaries set by the supervisor, since the supervisor expresses all allowable routes through the system, satisfying the specifications.

To set the scene, we will begin with a brief introduction of the concepts to be presented in detail. This introduction comes in the form of an example system modeled by the object-oriented principles described in Chapter 5, with a supervisor synthesized by the variation of the supervisory control theory that will be presented in Chapters 3 and 4. First the example system is described, followed by a short description of the object-oriented modeling approach in general, and its application to the example system. Then some aspects of distributed product specification is discussed, and two products to be produced by the example system are introduced. Finally, the SCT algorithm is briefly described, and applied to the example system. The presented example will by no means be exhaustive, but we feel that this introduction gives a reasonable indication on how object-oriented modeling and the SCT can be used in practice, and, thus, motivates the remaining chapters of this thesis.

## 1.1 An Example System

The example assembly system is shown schematically to the right in Figure 1.1. It consists of an input buffer, a lathe, a mill and an assembly unit. The system also contains a robot for loading and unloading the production resources. These are the resources that we will regard here. The operator station for manual supervision of the production, and the bar code reader that registers incoming workparts will not be considered.

The input buffer, **M1**, has a limited capacity of one product at a time. Products are loaded onto **M1** on request, by some external source which is not modeled and is considered to be of infinite capacity. From **M1** the products are transported by the robot to either the lathe, **M2**, or the mill, **M3**, depending on the specification of the product route. The assembly unit, **M4**, takes two different products—loaded one at a time—

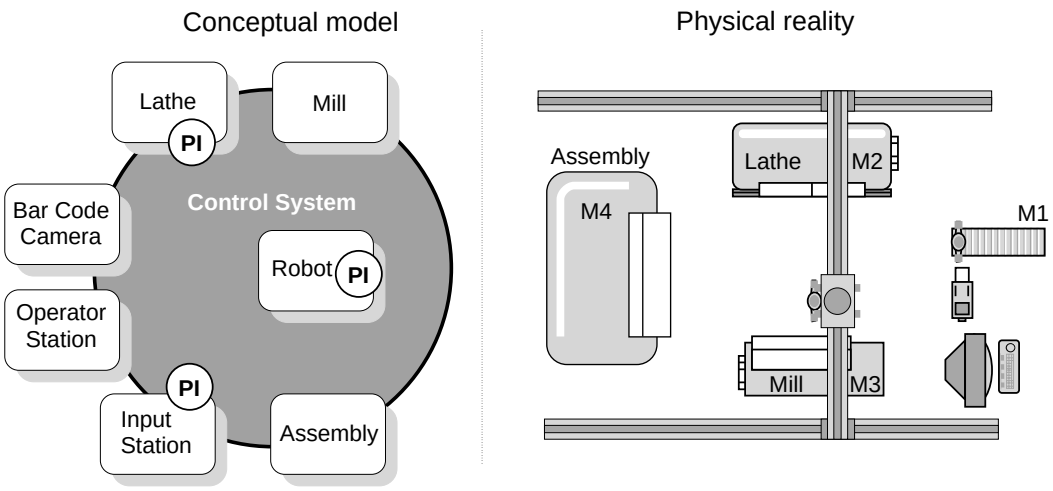



**Figure 1.1:** Object-oriented model of a flexible assembly cell. The PI's represent product route specifications.

assembles them and emits one product to some external sink outside the system. As with the external source, the external sink is not modeled, and considered to be of infinite capacity. **M2** and **M3** can both handle only one product at a time.

Note that this is a purely imaginary system, carefully chosen so as to illustrate the core of the concepts described in this work. However, the example system has many things in common with real life flexible manufacturing and assembly systems.

## 1.2 Object-Oriented Modeling

For the generation of any type of control system, a model of the system to be controlled is needed. It is favorable for this model to lie as close to the physical representation as possible, yet encompass as little information as absolutely necessary. A manufacturing system involves a number of independent manufacturing devices interacting to perform useful work. Object-oriented modeling caters for abstraction by encapsulating data and behavior. It also modularizes the modeled system in a natural way by resulting in an object structure close to the physical structure. The objects are independent software modules interacting by messages.

### 1.2.1 Internal Resources

An internal resource is an autonomous reusable software model, described as a DEP. The reusability emanates from the fact that machining devices can be described by their general behavior on an abstract level. On this level, application-specific details are not visible; a crucial requirement for the generation of reusable software components. However, the functionality promised by the abstract level, the *general part*, has to be implemented somewhere. Therefore, each resource also has a *specific part* that interacts with the general part. This specific part is tailored to the requirements of the actual physical device

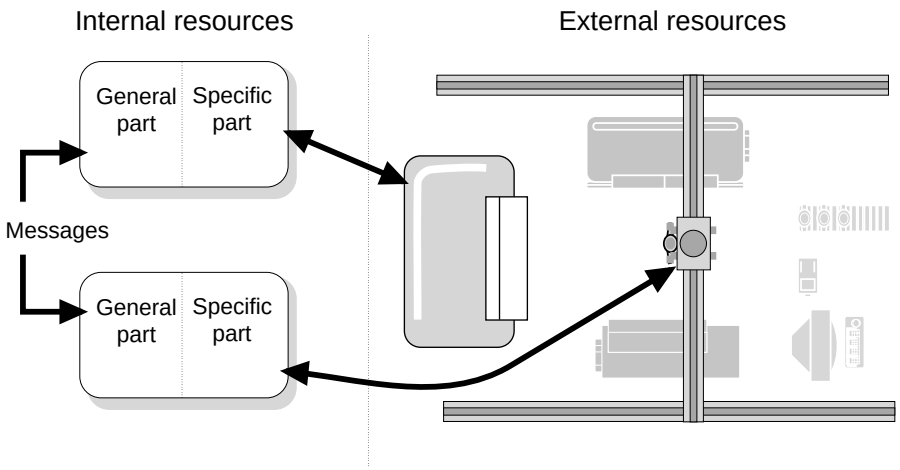



**Figure 1.2:** Each internal resource consists of a general and a specific part, thus separating the control of the assembly process from the control of the individual devices.

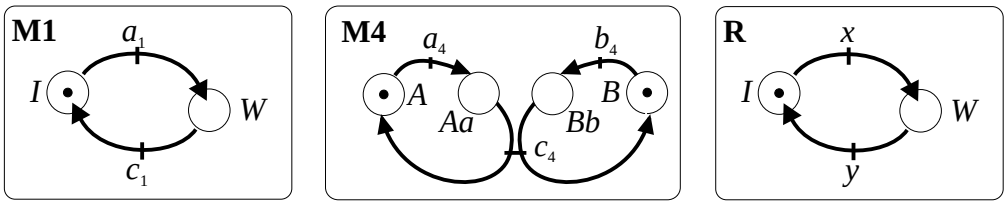



**Figure 1.3:** **M1** represents a general production unit, such as a lathe or a mill. **M4** represents a general assembly unit, and **R** represents a general transporting device.

in the specific application. Thus, control over a physical device is routed through the corresponding internal resource, see Figure 1.2.

Operating concurrently, the internal resources constitute the plant to be controlled. This concurrent execution is modeled by *full synchronous composition* (FSC) of the internal resources, see Hoare (1985). This allows the internal resources to be coupled and thus able to synchronize, for instance, over mutual use of tools.

General parts, representing generic production resources such as a milling device and an assembly unit, are given in Figure 1.3; in this case in the form of Petri nets, see Peterson (1981). These models include only details specific for supervisor synthesis. More elaborate resource models usable for actual control can be found in Gullander (1995) and in Section 5.2. Suffice it to say that the models presented in Figure 1.3 are subsets, in some sense, of the more elaborate reusable resource models of Section 5.2. We can also note the work of Tittus (1995b), were reusable resource models are given for control of chemical batch processes.

The resource models of Figure 1.3 simply model whether a resource is available or not. The model of **M4** being somewhat more complicated, since it includes the fact that two products have to be loaded, before an assembly operation can take place. The details of the actual operations to perform, or that can be performed, are not included in the

models. Once a product has claimed a resource, the actual operation is of no concern to the supervisor. The supervisor merely coordinates the dynamic resource allocation among the products.

## 1.3 Distributed Product Specification

Products are also modeled as DEPs. Each product specification describes a number of alternative desired routes through the system. It is thus natural to view a product as a specification on the manufacturing process to exhibit a certain event-sequence. However, there can be several independent products using the plant simultaneously. Together these form a joint specification on the system's overall behavior. So as not to overburden the user with an overly detailed knowledge of the system, it is important that there exists support for *high-level distributed* specification of the product routes.

### 1.3.1 High-Level Product Specification

By a *high-level product specification*, we mean that the specification of the product route does not explicitly mention specific devices pertaining to a particular application. Rather, the product route is given as an *operation list* (see Andréasson (1995) and Section 5.3), a sequence of operations that the product has to undergo to be produced. In this way a high-level product specification is independent of any plant that is to manufacture the product. Any plant that offers the required operations can produce the specified product. Compare with the *general recipe* of Tittus (1995b).

The preferred way to specify operation lists is graphical. The operator lays out the desired operation list, focusing merely on what operations to perform and in which order. A graphical layout is well suited for a computerized tool with a graphical interface. A number of graphical high-level operators for specifying operation lists and product routes are given in Section 5.3.1. High-level operators in an algebraic form are shown in Andréasson (1995).

However, once a plant is chosen to manufacture the product, the specified operations can be mapped onto the capabilities of specific resources within that plant. This mapping procedure is described for batch processes by Tittus (1995a), and briefly exemplified in Example 5.2 on page 156. The result is a specification for the product consisting of all possible sequences of resource allocations that will manufacture the product. Note that this is still a distributed specification. No regard for other products that may be simultaneously produced by the plant is taken. Such a product specification will be called a *product route* in this work, while it is called a *master recipe* by Tittus (1995b). Naturally, a product route can be manually specified for a given product and plant

For the example system two operation lists are given to the left in Figure 1.4. These are overly simplistic, for instance, no alternative paths are specified. Nonetheless, they serve well as an example.

The two product types, **AB** and **CD**, are assembled from **A** and **B**, and **C** and **D** parts, respectively. Thus, **A**, **B**, **C** and **D** represent raw material entering the system at **M1**, while **AB** and **CD** represent the finished products. The **C** and **D** parts are not processed upon, other than in the assembly unit, **M4**, while the **A** and **B** parts are

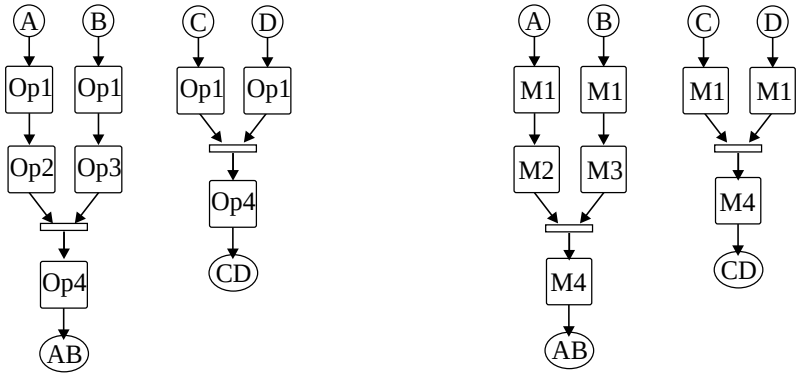



**Figure 1.4:** To the left is shown operation lists for two types of products to be manufactured by the example system. To the right is shown the corresponding high-level product routes, assuming that operation **Opi** can be performed by machine **Mi**, only ($i = 1, 2, 3, 4$). Note that both product types use **M1** and **M4**.

operated upon by the lathe, **M2**, and the mill, **M3**, respectively. The exact specifications for these operations are given as parameters "hidden" within the boxes of Figure 1.4. Note that these parameters can represent an entire program for a numerically controlled production resource, if necessary. However, for the control of the overall assembly process, the nature of the operations being performed within each resource is not important, merely the sequencing between the resources is regarded.

The operation lists of Figure 1.4 can, together with a model of the plant, be translated into product routes that point out specific resources of the given plant. These are shown to the right in Figure 1.4. These are not yet DEPs, since they only describe the available sequences of resources that the products are to visit. A detailed example of this, relating to chemical batch processes, is shown in Tittus (1995d). This example is also briefly discussed in Example 5.2 on page 156. The approach is equally appropriate for manufacturing and assembly systems.

For the synthesis of a supervisor, we need the product routes in a "lower-level" form of DEPs. The resulting "low-level" product routes are given in Figure 1.5. Since there is a well-defined correspondence between a product route given as a DEP, as in Figure 1.5, and given in a more "high-level" way, as in Figure 1.4, we will make no clear distinction between the two. When necessary we will speak of "high-level product routes". Note though, that Tittus (1995d) does distinguish between these two views of a product route; the "high-level" one is called a *master recipe*, while the "low-level" DEP is called a *synchronizable master recipe*. The indices of the events denote the respective resources, see Figure 1.3.

We can note that, though the robot is required for moving products between the machines, it is not modeled in Figure 1.4, neither for the operation lists or the high-level product routes. This is not needed, since there is only one robot that is assumed to be able to handle all products and serve all machines. If this was not the case, there were multiple robots each of which could only handle specific types of products, for instance, then these would have been included in the high-level product routes. However, note that the events pertaining to the robot, $x$ and $y$, are present as elements of the ordered pairs,

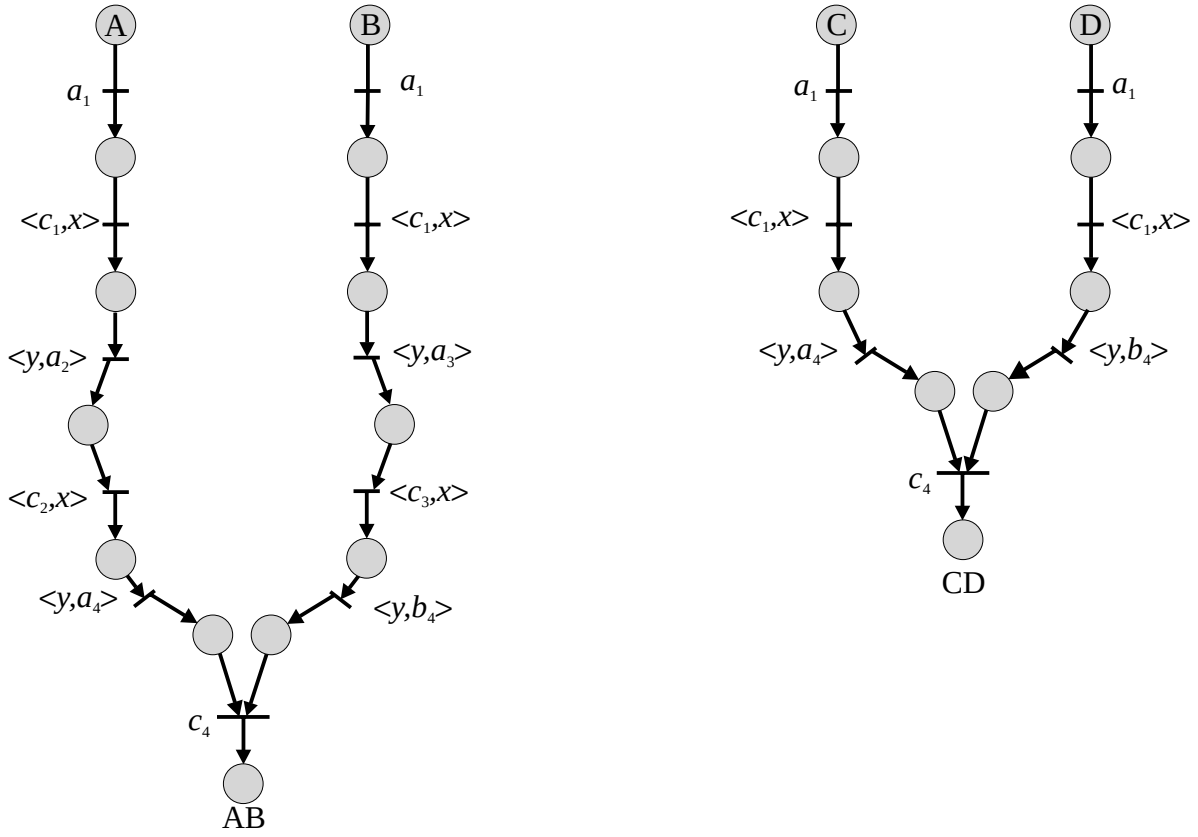



**Figure 1.5:** A Petri net describing the high-level product routes of Figure 1.4 in conjunction with the resource models of Figure 1.3.

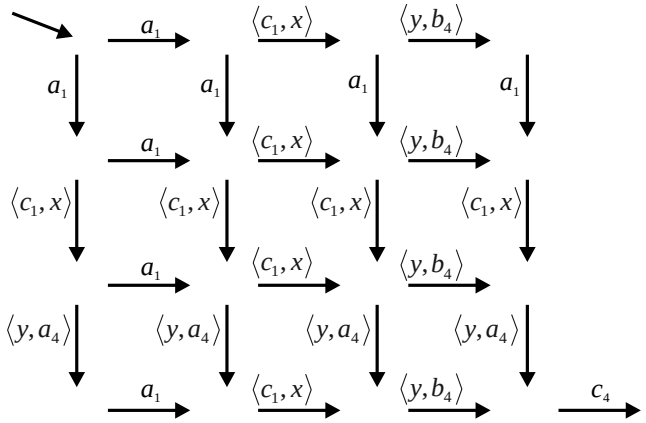



**Figure 1.6:** The state-machine representation of the product route for the **CD** product of Figure 1.5, assuming one part each of **C** and **D**. The initial state is indicated by the unlabeled arrow to the top left. Note that this state-machine is non-deterministic, since the $a_1$ event leads from the initial state to either of two states.

$\langle c_1, x \rangle$, for example. The reason for these ordered pairs is explained in Section 1.4.2.

### 1.3.2 Interleaving

A *distributed product specification* means that all products can be specified independently of each other. The operator laying out the operation list does not have to consider any other product that may be present within the system simultaneously. Only the specific product that is being specified is of concern, even though there may be a multitude of different products running concurrently through the same system.

The product routes describe claiming and releasing of resources that constitute the plant. Claiming a resource and releasing it modeled as events, so that the product routes have event sets that are subsets of the plant's event set. Thus, it is not uncommon for different product routes to have common events, on the contrary. In Figure 1.5 we can see that both product routes have the events of **M1** and **M4** in common. At the same time, for maximal utilization of the plant, all product routes must be able to run as unconstrained by all other product routes as possible. Given distributed product specifications, a joint specification on the overall systems behavior is obtained by composing the independent product routes by *interleaving*.

The concept of interleaving is described by Hoare (1985). Essentially, interleaving means that two DEPs can execute their events asynchronously and irrespective of each other, even though there may exist mutual events. In fact, interleaving explicitly prohibits synchronous execution of any event.

Interleaving is inherent in the asynchronous execution of two or more DEPs. At each time instant any DEP can execute its own event, asynchronously with regard to any of the other DEPs. Thus, at each time instant the total system behaves as either of the DEPs, and at no times will two DEPs engage in the same action synchronously. Furthermore, the execution of an event is considered to be an instantaneous atomic action, without duration. Therefore, no two different events, equally labeled or not, can occur simultaneously. Thus, the event sequences of the interleaved system is the interleaving of the event sequences of the respective DEPs. Figure 1.5 shows the interleaving of the distributed product specifications of the example system in Petri net form. The interleaved product specification of all simultaneously present product routes will be denoted *Sp*, and is said to be a *local specification*.

Each product route defines a number of states that the system is desired to be able to reach. The completion of a product is typically such a desired state. Thus, the product routes describe *desired paths* through the system. These desired paths are naturally retained in *Sp*. However, in *Sp* there may also exist states that we do not want the system to be able to reach. These states can be designated as *forbidden*. With multiple robots, such a state may be that one robot unloads a machine, while at the same time another robot is to load that machine. Furthermore, some paths through *Sp* may be either desired or forbidden. This can also be introduced by synchronizing *Sp* with some auxiliary specification expressing these paths. See Lin (1988).

Due to the fact that multiple product routes may have equally labeled simultaneously executable transitions, the local specification may be *non-deterministic*, in the sense that one and the same string of events can lead to any of a number of states. This non-determinism arises naturally, given a number of independent resources and distributed

specifications of products to be manufactured by these resources. However, this non-deterministic specification is also an added complexity with regard to the SCT, which originally only considered deterministic plant and specification. In the following chapters will be shown that the SCT can be extended to handle the case of non-deterministic specification.

We can also note that the individual **A**, **B**, **C** and **D** parts use the same resource **M1** when entering the system. Thus, not only may non-determinism arise due to the interleaved product routes as such, but within each product route the respective parts may also be interleaved, and so the product routes themselves are non-deterministic. In Figure 1.6 the product route for the **CD** product is given in state-machine form, making the non-determinism more explicit.

## 1.4 Supervisory Control

A *supervisor* is a DEP that operates in synchrony with the plant, influencing the plant so as to have the closed-loop system of plant and supervisor exhibit some pre-specified desired behavior. For our purposes, the full synchronous composition (FSC) adequately models the interaction between the plant and the supervisor. When synthesizing the supervisor, the SCT regards the plant as a *generator* of events; all events occur as a consequence of some action within the plant. Thus, the supervisor is confined to restrict the actions of the plant by *disabling* events as the system executes. With the FSC this disabling is a matter of the supervisor not defining some disabled events at each closed-loop system-state. Note that in different closed-loop system-states different events can be disabled.

However, not all events generated by the plant can be disabled by the supervisor. The set of events of the plant is partitioned into two disjoint event-sets, the *controllable* and the *uncontrollable* events. The controllable events can be dynamically disabled by the supervisor so as not to occur. The uncontrollable events are, on the other hand, not subject to influence by the supervisor; the plant can generate any of these whenever it occupies a state from which an uncontrollable event is valid. The completion of an operation is a typical uncontrollable event; once the supervisor has allowed the operation to start (a controllable event), the supervisor has no control over when the task is completed. A machine breakdown, is another typical uncontrollable event[1]. For the supervisor not to become "out of sync" with the plant, it is imperative that the supervisor is able to follow all uncontrollable events that can be generated in each closed-loop system-state; the supervisor must be *complete* with respect to the plant.

It may be desirable for the closed-loop system to always be able to reach some significant state. For instance, the state denoting completion of production with all products satisfactorily manufactured is a typical desired state. For this, the specification includes *marked* states. The closed loop system must be such that from any state reachable from the initial state, some marked state can always be reached. This ensures that the system will never be controlled into a state from which further production cannot continue. A DEP that is such that all states can reach some marked state is said to be *coaccessible*. Furthermore, to retain only states that can in practice be reached, we will require the

[1] One which we have chosen not to model, see the resource models of Figure 1.3.

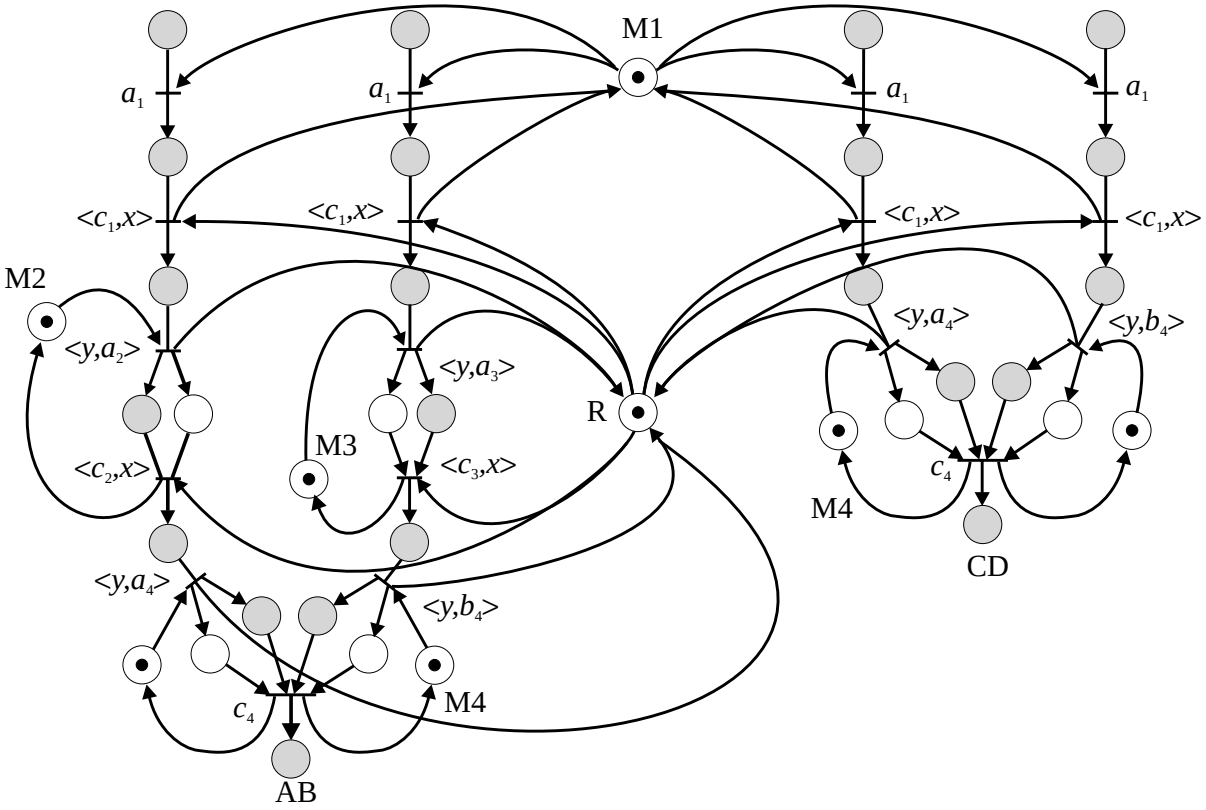



**Figure 1.7:** The global specification $P \| Sp$ expressed as a Petri net. Note that, for clarity, the places corresponding to **M4** have been duplicated, and places corresponding to **M1** and **R** have been removed.

closed-loop system to be *accessible*. A DEP which is both accessible and coaccessible is said to be *trim*. When the closed-loop system is trim, any event sequence can be continued into a marked state, ensuring that the specification can always be met.

### 1.4.1 Global Specification

The local specification $Sp$ contains all possible interleavings of the desired product routes possibly with other specifications for forbidden or desired states and forbidden paths. Not all of these interleavings are physically possible, though, due to the configuration of the plant, $P$. In fact, $Sp$ does not fully describe the desired behavior of the plant. This is so, because some events of $P$ may not be in the event set of $Sp$, and thus the plant can execute any of these whenever in a state to do so.

To retain only the physically possible and desired routes, $Sp$ is synchronized with the plant under full synchronous composition. That is, the *global specification* $P \| Sp$ is generated, see Figure 1.7. This guarantees that only physically possible routes are expressed, and it also guarantees that the desired behavior, expressed by $P \| Sp$, is a restriction of the possible behavior, expressed by $P$. However, in the global specification some combinations of product routing will inevitably block the system from ever reaching a state where all products have been satisfactorily completed. For instance, in Figure 1.7

it is clear that allowing two consecutive **C** parts to enter the system will indefinitely prohibit the assembly of any product.

Furthermore, the uncontrollable events of $P$ may have as a consequence that not all of the behavior expressed by $P \| Sp$ is controllable. Some states of the global specification must not be reached, otherwise some uncontrollable event may take the plant to some undesirable state, not allowed by the specification. Some combinations of product routing must be prohibited, and this is one of the tasks of the supervisor.

### 1.4.2 Event Connection

The algorithm presented in Chapter 4 concerns finite state automata. Therefore, the (finite) state automaton representing the DEP $P \| Sp$ must be generated. This is the reachability graph of the global specification, the Petri net of Figure 1.7. Since $P \| Sp$ is a bounded Petri net, this can be done, see Peterson (1981). Naturally, it requires an initial marking vector to be specified for the Petri net. The number of states of $P \| Sp$ is considerably smaller than the number of states of $Sp$, primarily because the number of physically possible routes through the plant is limited. This is one of the reasons for first using some "reduced state" description of DEPs, like Petri nets, to express $P$, $Sp$ and $P \| Sp$, and then generating a state automaton representation of $P \| Sp$.

Some transitions of Figure 1.7 are labeled by ordered pairs of events from $P$. These transitions describe the passing of the product from one resource to another, and consists of the "exit" event of the former resource together with the "entry" event of the next. These ordered event-pairs arise when generating the DEP representation of the product routes, and can be automatically introduced. This models the fact that when passing a product from one resource to another, the product actually occupies *both* resources simultaneously. Thus, no other event can be allowed to happen in-between the two events of such an ordered pair. We say that the events are *connected*. This means that we regard the connected events as *one* event, even though the individual events are named differently.

Not including connected events would mean that the model of the system allowed a product to "exit" **M2**, say, without "entering" any other resource. The robot **R** could be used to load **M2** from **M1**, even though the product cannot "exit" **M2** without the robot being present.

Event connection can be seen as a *product-unspecific* specification, not dependent on the product routes but on the physical connectivity of the plant. Event connection is a consequence of the fact that we have autonomous resources with mutually disjoint alphabets. Between any type of resources shared by the same product route, event connection is required . Without event connection the resources would have to be explicitly coupled by common events. This would hinder their reusability and the flexibility of the system. The number of required "exit" events, and their labels, of one resource would have to match the number of resources that it is to be "connected" to, as well as having equal event labels. Thus, introduction of a new resource would require altering other resources. The mere instantiation of an appropriate object would no longer be possible when introducing a new resource into the system. Compare the approach of Banaszak (1990), where each resource is modeled specifically for the products it is to operate upon.

In the state automaton representation of $P \| Sp$, the connected events are "unfolded",

so that the global specification includes the same events as the plant. In Section 5.4.2 this is shown to be valid, since it generates a *subprocess* of $P \| Sp$ from which a valid supervisor for $P$ can be synthesized.

### 1.4.3 The Supervisor

In the state-machine representation of the global specification, arbitrary transitions and states can be removed; states and transitions that are considered unwanted during execution. For instance, two resources may not be allowed to work simultaneously, because of constraints on the amount of power drawn. In such a case, all states representing those two resources working simultaneously can be removed. Removal of the specified states and transitions, results in the *final specification*, within which the supervisor is to control the plant.

The synthesis of a supervisor is an iterative process. Except for special cases, see Brandt (1990), no closed form expressions exist. In the approach described in this work, a supervisor is a *subprocess* of the final specification. A subprocess is a DEP with its graphical structure contained within another process, the *superprocess*. A subprocess can be generated from a given process by removing states and transitions between states. In generating the supervisor, a complete and trim subprocess of the final specification is synthesized. Naturally, there may exist several solutions to the given problem of finding a complete and trim supervisor for a given plant. However, an additional requirement is to find the supervisor allowing the largest possible behavior, that is, the supervisor is required to be *minimally restrictive*. Since the supervisor is an exact model of the plant under supervision, it is clear that the minimally restrictive complete and trim supervisor is the *maximal complete and trim subprocess* of the control recipe. A formal description of the supervisor synthesis algorithm will be given in Chapter 4.

Usually the number and types of resources are constant and known, while the number and types of products is time-varying. New products are initiated and old ones terminate, asynchronously and independently as the work progresses. Thus the total number of products, and their routes through the system cannot be known *a priori*. For the sake of flexibility we must allow any possible route through the system. We must also allow inclusion of new products at any time during execution. However, with the introduction of new product routes there may arise additional constraints on how the products can be allowed to run concurrently. Thus, we start up the system with a number of initial product routes and calculate a supervisor for these. When new orders for more products or new product types arrive, we interleave the new routes in their initial-states with the old routes in their *current* state. The resulting joint specification is then composed with the plant in its current state, and a new supervisor can be calculated. Even as product routes terminate, this calculation can be performed so as to minimize the size of the supervisor.

The supervisor is a state automaton expressing all physically possible and allowable routes through the system, given the final specification. It is within the boundaries set by the supervisor that the system will be driven by the controller.

### 1.4.4 The Controller

We have carefully avoided defining the *controller*, mentioned above. This is for the reason that there are many ways to interpret the controlling entity, and they all come down to the question of event generation. Who sends which message when and to whom? That is, who generates which events and who follows?

The SCT normally regards the plant as generating all events. The supervisor merely follows and restricts the event generation. Thus, the plant and the supervisor operates in synchrony. For the supervisor and the plant to never get "out of synch" it is imperative that the supervisor can follow all events it allows the plant to generate. Restriction of an event is enforced by the supervisor not defining that event in the closed-loop system state. The uncontrollable events cannot be restricted from being generated by the plant. Therefore, it follows that the supervisor must be able to follow all uncontrollable events. It must be *complete* with respect to the plant. This is the philosophy we have adopted in generating the supervisor.

By Balemi (1992) the *controller* is regarded as a supervisor that generates some events, the *commands*, while the plant as generating other events, the *responses*. It is shown by Balemi (1992) that the commands can be equated to the controllable events, while the responses are equal to the uncontrollable events. Furthermore, Balemi (1992) shows that the controller and the plant have to be *mutually complete* so that the controller only generates events for which the plant is ready. In our case $S$ is derived from $P$ since it is the maximal complete and trim subprocess of the final specification. This means that the generated supervisor and the plant are always mutually complete. Therefore, we can regard the supervisor as a controller; an active entity controlling the plant by generating commands and receiving responses. The closed-loop system is then modeled by the synchronous composition of the plant and the controller. It will be shown, see Chapter 3, that this is in fact equal to the supervisor itself when the supervisor is complete (which it is) and the plant is deterministic (which it is). Thus, the supervisor is an exact model of the plant under supervision.

The supervisor coordinates the usage of the resources, commanding the robot to load and unload the machines as appropriate. At times the supervisor expresses non-deterministic choices of products to load or unload. The optimal choosing between such non-deterministic routes, taking into account aspects like fairness, due dates, etc., is a complex task, the implementation of which is outside the scope of this thesis.

## 1.5 Objectives of this Work

The goal of this work is to show how the supervisory control theory can be combined with an object-oriented modeling approach as applied to, for instance, flexible manufacturing systems. In doing so we primarily extend our previous work on object-oriented modeling of DEPs, mainly presented in [3]. It has been shown by Adiga (1993), Joannis (1992) and Jobling (1994), as well as by others, that object-oriented analysis and modeling caters for the partitioning and structuring of a large problem domain into manageable pieces. The main benefits of object-oriented modeling of manufacturing systems comes from using reusable software modules. This has a great impact on the structuring, modularization and implementation of the system. There arises a natural separation of the control of the

individual subsystems from the synchronization of the system as a whole. This immediately brings about the possibility for distributed high-level product specifications, as was shown in the previous sections.

### 1.5.1 Contributions

The objective is now to show that the supervisory control theory can be adapted to systems with the above properties. The main contributions of this work is then the following:

- We introduce an object-oriented modeling approach for fabrication processes, building on the ability to extract general behavior into reusable models of the physical devices; general in providing functionality common to all devices of a similar class, and reusable from a viewpoint of extendibility. A more elaborate description of this modeling approach is given in references [1], [2] and [3].

- The outcome of that modeling approach is set of internal resources, that constitute the plant to be controlled. These describe the capabilities, restrictions and dynamic behavior of the physical devices. Generic models in the form of discrete event processes will be given.

- We show that for such systems, the specification on the desired behavior can be given as a set of operation lists describing the desired sequences of general operations for production of the respective product.

- The internal resources and the operation lists emphasize the separation of the control of the individual devices from the overall synchronization of the fabrication process. We maintain that this is a crucial issue for reusability and true flexibility.

- Based on the operation lists we generate product routes describing desired routes through the system. The concurrent execution of the product routes is modeled by interleaving so that the joint specification of all products simultaneously present within the system is typically non-deterministic.

- The supervisory control theory of Ramadge (1987) and Wonham (1987) is extended to encompass non-deterministic specification. It is shown that the notion of controllability is not strong enough to guarantee a complete supervisor.

- We give algorithms for synthesizing a complete supervisor such that the closed-loop system is always nonblocking, given a non-deterministic specification.

- We also show that the input/output formulation of Balemi (1992) is valid for this case, so that the supervisor itself can act as an active controller, driving the plant within the given specification and outside the undesired states.

### 1.5.2 Assumptions and Restrictions

In this thesis we will make the following assumptions and restrictions.

- We will only regard finite state automata, and hence, regular languages.
- We will neglect the combinatorial explosion of the number of states. Automata is used merely as a formal tool for proving our statements and algorithms. In practical implementations efficient approaches like binary decision diagrams, for instance, will probably have to be employed.
- We will disregard the *silent event*, so that non-deterministic choice will only be modeled by equally labeled transitions emanating from the same state.
- The plant that we control will be regarded as non-marked. Marking is introduced as a means of specification.
- We will always assume that the event-set of the supervisor is equal to the set of events defined by the plant. Thus, the supervisor expresses *all* desired and allowed behavior. The plant has no additional freedom in generating events not in the event-set of the supervisor.
- All events are considered to be observable. Non-determinism is not a consequence of un-observable events.

Note also that we elaborate on the applicability of the supervisory control theory for non-deterministic specification as well as non-deterministic plant. However, the application of the presented algorithm is restricted to the case of a deterministic plant.

### 1.5.3 Thesis Outline

This work is organized as follows. A thorough definition of the *finite transition machines* that we will use to model DEPs, is given in Chapter 2. A number of properties are shown, and the special case of two transition machines, with one being deterministic and the other being non-deterministic, is investigated. It is shown that when the non-deterministic transition machine holds the property of *refinement* relative to the deterministic one, then their synchronous composition will be equal to the non-deterministic transition machine. The definitions and results presented in Chapter 2 are essentially well-known, though the notion of refinement is extended to non-deterministic processes. The definitions of operations on subprocesses seem to be novel. In the literature the state-spaces are required to be disjoint; see Eilenberg (1974) and Hopcroft (1979), for instance.

In Chapter 3 we first restate the original supervisory control problem, as formulated by Ramadge and Wonham. This is given complete with proofs of the propositions, significantly simplified compared to the original proofs. A more "modern" notation has been used, mainly by introducing the full synchronous composition as modeling the interaction between the supervisor and the plant. This has already been done in Kumar (1991), but the treatment there did not include all of the SCT; marked states were not treated, for instance. We feel that the presented notation casts some light on the SCT for the unfamiliar reader, as well as helping the reader familiar with the SCT in its original form to better understand the extensions that follow. The *derivation* of the controllability property is, as far as we know, our own. Wonham (1987), Ramadge (1987), Balemi (1992) and Kumar (1995), among others, *define* controllability but do not *derive* it.

Since the input/output formulation of Balemi (1992) lies closer to a control engineering point of view, this approach is also described in Chapter 3. The inverse properties are not our own, though the terminology is. These properties are defined by Balemi (1992), though he makes no distinction between controllability and completeness.

Then follows the application of the SCT to the general case of non-deterministic transition machines in Section 3.4. It is shown that the notion of controllability is no longer strong enough to guarantee a usable supervisor. Necessary and sufficient conditions are given for a non-deterministic supervisor, $S$, to be complete with respect to a plant, given a controllable language, $L(S)$. Furthermore, it is shown that the input/output formulation is also encompassed by the non-deterministic supervisory control approach as given in this work. The notion of a nonblocking closed-loop system is investigated for the non-deterministic case. It is shown that a nonblocking supervisor is not enough to guarantee that the closed-loop system is nonblocking when non-determinism is present. Necessary and sufficient conditions for the marking of the supervisor relative to the plant is given, so that the closed-loop system is nonblocking. Finally, an overall view is given of a number of combinations of non-deterministic plant, specification and supervisor, with necessary and sufficient conditions for the existence of a supervisor.

Section 3.4 is almost entirely our own. Most of the results seem to be non-existent in the literature. The generalized definition of completeness has also been given by Overkamp (1994), though the generalized condition for a non-deterministic supervisor with a controllable language to be complete is our own. The extension of the input/output interpretation to non-deterministic systems has not appeared anywhere else, as far as we know. Also, to our knowledge the conditions for a nonblocking closed-loop system is not previously known. The theorems presented in the last section is not known to have been given in such a unified framework.

In Chapter 4 an algorithm for synthesis of a non-deterministic supervisor is derived. This is shown for a non-deterministic specification and deterministic plant. It is shown that the set of usable supervisors the specification defines relative to the plant can be ordered in an upper semilattice structure. From this, we show how to calculate the minimally restrictive supervisor such that the closed-loop system is nonblocking. That is, the supremal complete and trim subprocess of the specification.

Some of the lattice results given are well-known and given by Tremblay (1987), for instance. Other results for the specific application of calculating a complete and trim supervisor are new. The definition of the *supremal operator* to calculate the supremal element of the intersection of two upper semilattices, when it exists, is new. The algorithms given for calculating complete, accessible and coaccessible subprocesses are not new, however. Eilenberg (1974) also shows the algorithm for accessibility and coaccessibility, while the algorithm for completeness is well-known within the supervisory control theory. See Kumar (1991), for instance. What is new, is the application to non-deterministic systems, and the short and concise notation of the algorithms resulting from the definition of the various operators.

The object-oriented modeling approach that initiated this work, is briefly described in Section 5.2. This approach is described in detail by our previous work, as shown on page ii. Of greater importance to the present work are the various aspects of specification that are discussed, and the application of the mathematical framework of the previous chapters to the specific class of problems that arises. It is shown that the properties that have been

defined as necessary for the non-deterministic supervisory approach are immediate by the modeling approach. Thus, it is shown that the non-deterministic supervisory approach can be combined with object-oriented modeling techniques, and so we have a powerful framework for implementing large and complex discrete event systems.

A number of application examples are finally described in Chapter 6. The modeling and the synthesis of supervisors are shown to some extent.

# Chapter 2

# Finite Transition Machines

This work concerns control of what is usually known as *discrete event processes*, acronymed DEP. A discrete event process is a system that at each time instant occupies a *state* of being, one out of a finite number of possible states. The state has a symbolic value, rather than a numeric, and represents a situation of the process during which certain conditions hold. The DEP *transits* between the states according to the occurrances of *events*, occurring instantaneously at discrete intervals of time. Thus, the process' *behavior* is given by the sequences, *strings*, of events that occur, and the sequences of states that the process occupies due to these strings. These notions will be formalized in the following definitions.

Let us first, however, note that the term *finite transition machine* will be used, instead of the more common *finite automata* (see Eilenberg (1974) and Hopcroft (1979)) or *finite state-machine*. The reasons for this are similar to those given in Arnold (1994) for using the term *finite transition system*. Automata can be considered to be the fundamental concept for the formal description of DEPs. The term *transition system* is used by Arnold (1994) to distinguish automata viewed as formal systems containing states and transitions from automata regarded as machines to recognize certain languages. Also, finite state-machines normally define one or more initial states and a set of final states, see Eilenberg (1974). This is not necessarily so for the finite transition systems of Arnold (1994). See also Cassandras (1993) who gives good reasons for not always demanding an explicit initial state, as well as disregarding a set of final states. We will use the term *transition*, for the same reason as given in Arnold (1994), that is, to point out that we do not merely regard the DEPs as recognizers of specific languages. However, we will also use the term *machine* to point out that we do indeed require a specific set of initial states. Furthermore, we will also allow, but not require, a set of *marked* states. These are not to be regarded as *final* states, though. Rather, these states can be regarded as states within which the system is allowed to rest. Some subtask may be completed and the system waits for more tasks to perform.

This chapter will lay the formal ground necessary to prove that a finite transition machine can be controlled to adhere to a prespecified behavior. We will formally define what we mean by a finite transition machine and its behavior. A universal set containing all finite transition machines will be defined to bring meaning to the operations on transition machines. A number of properties relevant for the control of transition machines will be proved, and we will define an ordering relation between finite transition machines.

Finally, some other approaches to modeling discrete event processes will be briefly discussed, and compared to the approach described in this work. Mainly, differences arise in the definition of equality of transition machines, and we will show why we feel that the strong equality relation that we define is necessary in the context of this work. Most other approaches give a much weaker definiton of transition machine equivalence, see Section 2.5.

## 2.1 Universe of Discourse

A finite transition machine is a system that occupies a distinct state of being, from which it can transit to another state on the occurance of an event. Thus, the fundamental concepts are states, events and transitions. Formally we have the following.

**Definition 2.1 States, Events and Strings**

Let $Q$ and $\Sigma$ be two finite sets. For simplicity we will require the elements of these sets to be *singular*, that is they are *not* themselves sets. Let $Q^+ = \bigcup_{i=1}^{\infty} Q^i$, where $Q^1 = Q$, $Q^2 = Q \times Q$ and $Q^{n+1} = Q^n \times Q$. Similarily, let $\Sigma^* = \bigcup_{j=0}^{\infty} \Sigma^j$. The elements of $Q^+$, will be called *states*, the elements of $\Sigma$ are referred to as *events*, and $\Sigma$ itself will be called an *alphabet*. The set $\Sigma^*$ is the set of all finite sequences of events of $\Sigma$. An element of $\Sigma^*$ is called a *string*. The string of no events, $\Sigma^0$, is called the *null string* and is denoted $\varepsilon$.

**Remark.** The set $\Sigma^*$ is called the Kleene closure of $\Sigma$, see Hopcroft (1979). Of course, the Kleene closure can be applied to any set. Note that, for two arbitrary sets $E_S \subseteq E_P \Leftrightarrow E_S^* \subseteq E_P^*$.

Observe that the null string $\varepsilon = \Sigma^0$ is an element of $\Sigma^*$, while the element $Q^0$ does *not* belong to $Q^+$.

Unindexed lowercase letters, from the greek alphabet or from the beginning of the latin alphabet, will be used to denote events. For example, $\sigma, \sigma', a, b, c$ will all represent events. Strings will normally be denoted by $s, s', s''$ and the like. Since $Q \subseteq Q^+$, the elements of $Q$ are also states. Lowercase letters from the middle of the latin alphabet, possibly indexed, will denote states. That is, $q, q', p_0, p_n$ all denote states. Furthermore, states of $Q^+$ not in $Q$ will be given as ordered tuples of the singular elements of $Q$, thus $\langle p_0, q_0 \rangle$ and $\langle p_i, q_j, r_k \rangle$ are two states of $Q^+$.

As a convention we will, following Eilenberg (1974), systematically confuse an element $x$ of any set with the subset of that set $\{x\}$ consisting of $x$ only. Furthermore, we will regard the sets $(X \times Y) \times Z$, $X \times Y \times Z$, and $X \times (Y \times Z)$ as identical. This is also according to Eilenberg (1974), but note that it is contrary to Tremblay (1987).

□

The definition of the set of states as being $Q^+$ may need some elaboration. Normally, a given transition machine will have a state set of $Q$. In fact we can even claim that the state sets of a number of given transition machines *define* the state set $Q$. We will define operations composing different transition machines into new ones. These operations all make the state set of the composed transition machine equal to the cartesian product of

the state sets of the original transition machines, so that transition machines generated by the composition operators will acquire state sets from $Q^+ - Q$. These operations are the normal way to compose given transition machines, and for these compositions to be contained within the universe of discourse, this universe must include the cartesian products over all singular states and any number of compositions, that is $Q^+$. We will only consider finite number of compositions, though.

The transitions of a finite transition machine will, for historical reasons, be referred to as *edges*, see Eilenberg (1974). Intuitively, an edge defines the state from which the transition is valid, the label of the occurring event and the state that the transition transits to. Because of this, it will not make any sense to distinguish between two edges defining transitions from the same state, labeled by the same event and transiting to the same state. Therefore, for a finite transition machine the set of all edges can be defined.

**Definition 2.2 Edges and the Edgeset**

Let $E = Q^+ \times \Sigma \times Q^+$. The elements of the set $E$ will be called *edges*, and consequently $E$ is called the (universal) *edge-set*.

**Definition 2.3 Finite Transition Machine**

A *finite transition machine* $P$, is described by a 5-tuple $(Q_P, \Sigma_P, I_P, M_P, E_P)$ where

$$\begin{aligned} Q_P \subseteq Q^+ : &\quad \text{the } state - space, \text{ the finite set of states} \\ \Sigma_P \subseteq \Sigma : &\quad \text{the } alphabet, \text{ the finite set of events} \\ I_P \subseteq Q_P : &\quad \text{the set of } initial \text{ states} \\ M_P \subseteq Q_P : &\quad \text{the set of } marked \text{ states} \\ E_P \subseteq Q_P \times \Sigma_P \times Q_P : &\quad \text{the } edge - set \text{ of } P \end{aligned} \tag{2.1}$$

**Remark.** Note that the state-space as well as the alphabet, and hence the edge-set, are all required to be finite.

The edge-set $E_P \subseteq Q_P \times \Sigma_P \times Q_P$ defines the transitions of $P$. That is, the ordered triple $(p, \sigma, p') \in E_P$, with $p, p' \in Q_P$ and $\sigma \in \Sigma_P$, means that there exists a transition from the state $p$ to the state $p'$ labeled by the event $\sigma$. The state $p'$ is then said to be *reachable* from the state $p$ via the event $\sigma$. Since both $Q^+$ and $\Sigma$ are required to be finite, so are $Q_P$ and $\Sigma_P$ and, hence, also $E_P$. This is why we speak of a *finite* transition machine.

Note also that an edge is equivalent to a *directed path* (of length one) as defined graph theoretically. □

In the following we will define, propose and prove a number of properties concerning finite transition machines. In doing so we will extensively make use of set union and intersection, cartesian product and other operations from set algebra. As noted by Eilenberg (1974), to bring meaning to the operations on the defined entities we first have to define the *universe of discourse*, see Tremblay (1987), within which all elements are contained.

**Definition 2.4 Universe of Discourse**

For any finite transition machine that we define, the *universe of discourse* is $Q^+ \times \Sigma \times Q^+ \times Q^+ \times E$. That is, for any finite transition machine $P$, we have that $P \subseteq Q^+ \times \Sigma \times Q^+ \times Q^+ \times E$.

**Remark.** Note that we will equate the terms *discrete event process*, *finite state automaton* and *finite transition machine*. These will all be considered to refer to objects of the same type, that is, objects within the universe of discourse defined above. Furthermore, for the sake of brevity, we will often refrain from using the whole terms, merely saying *process*, *automaton*, *transition machine* or even use the more common term *state-machine*. □

It is clear that a finite transition machine has the equivalent expressive power of a finite state-machine, see Hopcroft (1979). However, a finite state-machine is normally defined with a *transition function* defining its transitions instead of an edge-set. For notational convenience, it is sometimes more appropriate to use the transition function[1], $\delta$, in place of the edge-set.

**Definition 2.5 Transition Function**

The *transition function* of a transition machine $P$, is a mapping $\delta_P : Q_P \times \Sigma_P \to 2^{Q_P}$ that maps a state $p \in Q_P$ and an event $\sigma \in \Sigma_P$ into the set of reachable states, $\delta_P(p, \sigma) \subseteq Q_P$. The transition function can be extended into strings over $\Sigma^*$ as $\delta_P : Q_P \times \Sigma_P^* \to 2^{Q_P}$. With $\varepsilon$ denoting the null string of $\Sigma_P^*$ we have

$$\begin{array}{ll} \delta_P(p, \varepsilon) = p & p \in Q_P \\ \delta_P(p, s\sigma) = \delta_P(\delta_P(p, s), \sigma) & \sigma \in \Sigma_P, s \in \Sigma_P^* \end{array} \tag{2.2}$$

The transition function can also be extended to sets of states as $\delta_P : 2^{Q_P} \times \Sigma_P^* \to 2^{Q_P}$, as

$$\delta_P(Q, s) = \bigcup_{p \in Q} \delta_P(p, s) \quad Q \subseteq Q_P \,. \tag{2.3}$$

**Remark.** For an event $\sigma \in \Sigma_P$ for which $\delta_P$ is undefined at state $p$ we have $\delta_P(p, \sigma) = \emptyset$. This distinguishes events that are not feasible in the state $p$ due to the structure of the transitions machine, from events that are feasible but have the property of not changing the state. Though $\delta_P(\cdot, \cdot)$ is formally defined for $\Sigma_P$ only, the case that a transition machine will have to consider events not in its alphabet will arise when the transition machine is composed to run concurrently with other transition machines. Events in the alphabet of one of the machines but not in the alphabet of the other machine, is of no concern to the other machine. The other machine is physically incapable of participating in, or even noticing, the execution of the event. Thus, the only reasonable definition of $\delta_P(p, \sigma)$ when $\sigma \notin \Sigma_P$ seems to be $\delta_P(p, \sigma) = p$. Note that this implicitly includes a self-loop on all events not present in the machines own alphabet.

□

For notational convenience, we will, as introduced in Overkamp (1994), use the name of the transition machine itself to denote its initial states, when appropriate. We will then drop the subscript, that is

$$\delta_P(I_P, \cdot) \equiv \delta(P, \cdot). \tag{2.4}$$

[1] Though, strictly speaking, $\delta$ is not a function for transition machines, the common name *transition function* will be used in this work. Compare Hopcroft (1979).

**Remark.** The transition function of a process $P$ is related to its edge-set as $\delta_P(p,\sigma) = \{p' \in Q_P \,|\exists(p,\sigma,p') \in E_P\}$ and that the edge-set can be described by the transition function as $E_P = \{(p,\sigma,p') \in Q_P \times \Sigma_P \times Q_P \,|p' \in \delta_P(p,\sigma)\}$. Consequently $(p,\sigma,p') \in E_P \Leftrightarrow p' \in \delta_P(p,\sigma)$. □

**Summary**

In this section we have formally defined finite transition machines that will be used to model DEPs. A global universe, of which all transition machines are elements, have been defined to bring meaning to the operators and relations between transition machines that will be defined.

The definition of a finite transition machine is, of course, fundamental to the rest of the work presented in this thesis. Operations and ordering relations will be defined on transition machines. The universe of discourse merely sets the boundaries to which we will confine ourselves.

## 2.2 Traces and Tracesets

The edges of a finite transition machine can be concatenated into sequences describing the states visited according to the occuring events. Thus, the behavior of the transition machine can be defined in terms of sequences of states and events. For an edge $e$ to be "valid" in a given state, $e$ must have that state as an "initial component". The transition on the event represented by the edge reaches another state, from which the valid edges must again have that state as initial components. These concepts can be formalized as in the following definitions.

**Definition 2.6 First, Label, Last**

For any edge $e = (p,\sigma,p') \in E$, we define three functions,

$$\begin{aligned} first(e) &= p \\ label(e) &= \sigma \\ last(e) &= p' \end{aligned} \tag{2.5}$$

**Definition 2.7 Well-ordered**

Two edges $e_1, e_2 \in E$ are said to be *well-ordered*, denoted $e_1 < e_2$, if the last state of $e_1$ is equal to the first state of $e_2$. That is

$$e_1 < e_2 \Leftrightarrow last(e_1) = first(e_2). \tag{2.6}$$

**Definition 2.8 Trace**

A *trace* $t \in E^*$ is an ordered set of well-ordered edges that can be written as (for $e_i \in E$, $i = 1,\ldots,n$ for some finite$n$)

$$t = e_1, e_2, \ldots, e_n, \tag{2.7}$$

such that $e_i < e_{i+1}$ *for* $i = 1,2,\ldots,n-1$ and $n > 0$ is the length of $t$.

Note that, though $E^*$ contains all finite sequences of edges, well-ordered or not, when we write $t \in E^*$ it will silently be understood that $t$ is a *valid* trace, that is, $t$ is a set of well-ordered edges. In the same spirit we will also allow us to write $et \in E^*$ meaning that, for $t = e_0 e_1 \ldots e_n$, $e < e_0$.

The functions first, label and last of Definition 2.6 can be extended to traces as, for a trace $ete' \in E^*$,

$$\begin{aligned} first(ete') &= first(e) \\ label(ete') &= label(e)\ label(t)\ label(e') \\ last(ete') &= last(e') \end{aligned} \tag{2.8}$$

**Remark.** Note that, while first and last (when extended to traces) still generate states, label generates strings of events by concatenating the labels of the edges of the trace. □

For completeness we also define the *null trace*, $\varepsilon$, as the trace of zero length, defined for each state. Thus, each state is able to reach itself, and both $first(\varepsilon)$ and $last(\varepsilon)$ can result in any state of the considered transition machine. The label of the null trace is of course $label(\varepsilon) = \varepsilon$, the empty string. Using $\varepsilon$ for both the null trace and the empty string will not cause any ambiguity.

Two traces $t_1, t_2 \in E^*$ are well-ordered, that is $t_1 < t_2$, if $last(t_1) = first(t_2)$. Thus the null trace is well-ordered with respect to any trace (or edge).

**Definition 2.9 Prefix, Subtrace, Suffix**

For a trace $\tau \in E^*$ such that $\tau = t't t''$

$$\left.\begin{array}{c} t' \in E^* \text{ is called a } \textit{prefix} \\ t \in E^* \text{ is called a } \textit{subtrace} \\ t'' \in E^* \text{ is called a } \textit{suffix} \end{array}\right\} \textit{ of } \tau \tag{2.9}$$

**Remark.** For a trace $t \in E^*$, the prefix of length one is called the *initial edge* of $t$. The suffix of $t$ consisting of all but the initial edge, is called the *tail* of $t$. The null trace, $\varepsilon$, is a prefix to any trace. □

**Definition 2.10 Set of All Prefixes**

The set of all *prefixes* of a trace $t \in E^*$ is denoted $\bar{t}$ and is defined as

$$\bar{t} = \{t' \in E^* \,|\exists t'' \in E^*\ t't'' = t\}. \tag{2.10}$$

**Remark.** Of course, every trace is a prefix of itself and therefore $t \in \bar{t}$. □

**Definition 2.11 Similar, Related**

Two traces $t_1, t_2 \in E^*$ are said to be *similar* if they have the same label. That is

$$t_1\ sim\ t_2 \Leftrightarrow label(t_1) = label(t_2). \tag{2.11}$$

Two traces $t_1, t_2 \in E^*$ are said to be *related* if they reach the same state. That is

$$t_1\ rel\ t_2 \Leftrightarrow last(t_1) = last(t_2). \tag{2.12}$$

**Remark.** Traces that are not similar will be called *dissimilar*, and traces that are not related will be referred to as *unrelated* or *diverging*. Traces of different transition machines can, of course, be similar. However, it does not make sense to consider whether two traces of different transition machnes are related or not. Since two traces are related only if they reach the same state, this is obvious. One and the same state cannot belong to two different transition machines, even if there may exist equivalently labeled states in several machines. However, we will sometimes, with abuse of terminology, refer of traces of different transition machines as related. Note though that we mean by this that the traces reach equally *labeled* states, not equal states.

It is easy to see that, whenever two traces are dissimilar or unrelated, they are also nonequal. This also means that two equal traces are also similar and related. These are immediate consequences of the above definitions. However, note that the converse does not necessarily hold. Two traces can be similar and related without necessarily being equal. There exist certain conditions on the structural properties of a transition machine, under which the converse holds, though. Such as when a transition machine is *deterministic*, for instance. □

### 2.2.1 Tracesets

Since a transition machine $P$ defines a subset $I_P$ of its state-space $Q_P$ as initial states, not all well-ordered traces of $E_P^*$ are generated by $P$ as it evolves. That is, the behavior of $P$ does not include all traces definable over the edge set $E_P$, but merely those traces which start in some initial state. Thus, for a finite transition machine we define its (closed) traceset as the set of traces that can arise when all possible combinations of valid transitions beginning at $I_P$ are considered.

**Definition 2.12 Closed Traceset**

The (closed) *traceset* of a transition machine $P$ is the set of well-ordered traces beginning at initial states. This is defined as

$$tr(P) = \{t \in E_P^* \,|first(t) \in I_P\}. \tag{2.13}$$

**Remark.** The traceset $tr(\cdot)$ defines all traces beginning at some initial state. Therefore, for a trace $t \in tr(\cdot)$, all the the prefixes of $t$ are also included in $tr(\cdot)$. That is $\bar{t} \subseteq tr(\cdot)$. Therefore, when necessary for distinction, $tr(\cdot)$ is also called the *closed* traceset. □

In general, a transition machine also encompasses a number of marked states. It is then interesting to consider the set of traces beginning at the initial states and reaching the marked states. This set is called the *marked traceset*.

**Definition 2.13 Marked Traceset**

The *marked traceset* of a transition machine $P$ is the set of well-ordered traces beginning at initial states and reaching marked states, defined as

$$tr_m(P) = \{t \in tr(P) \,|last(t) \in M_P\}. \tag{2.14}$$

**Remark.** Since both the closed and the marked tracesets, $tr(\cdot)$ and $tr_m(\cdot)$, of a transition machine are defined from the initial states, and since $last(tr_m(\cdot)) \subseteq last(tr(\cdot))$, it is obvious that the marked traceset must be a subset of the closed traceset. □

For ease of notation, we will extend the functions of $first(\cdot)$, $label(\cdot)$ and $last(\cdot)$ to any set of traces, most notably to the traceset of a given transition machine. For an arbitrary set of traces, $\mathrm{T} \subseteq E^*$, we have that

$$\begin{aligned} first(\mathrm{T}) &= \bigcup_{t \in \mathrm{T}} first(t) \\ label(\mathrm{T}) &= \bigcup_{t \in \mathrm{T}} label(t) \\ last(\mathrm{T}) &= \bigcup_{t \in \mathrm{T}} last(t) \end{aligned} \tag{2.15}$$

Of course, for a transition machine $P = (Q_P, \Sigma_P, I_P, M_P, E_P)$, we have that $first(tr(P)) = I_P$ and $last(tr_m(P)) = M_P$. The set of strings given by $label(tr(P))$ will be defined to be the *language* of $P$ in Definition 2.21 of Section 2.2.3.

**Definition 2.14 Prefix Closure**

The *prefix closure* of a set of traces $\mathrm{T} \subseteq E^*$, is defined as

$$\overline{T} = \bigcup_{t \in T} \bar{t}. \tag{2.16}$$

**Remark.** For any traceset we have that $T \subseteq \overline{T}$. When $T = \overline{T}$ this is said to be a *prefix closed* traceset. Note that, for any transition machine the traceset $tr(\cdot)$ is always prefix closed, while the marked traceset $tr_m(\cdot)$ seldom is. It follows directly from Definition 2.13 that the prefix closure of the marked traceset is a subset of the closed traceset, so that the following ordering always holds,

$$tr_m(\cdot) \subseteq \overline{tr_m(\cdot)} \subseteq tr(\cdot). \tag{2.17}$$

. □

## 2.2.2 Accessibility

The structure of a finite transition machine may be such that some or all of its states can be reached from the initial states, and from some or all states some marked state can be reached. This is of importance when it is required to guarantee that the transition machine is *live*, that is, that some marked state can always be reached. When all states are reachable from the initial states, the transition machine is said to be *accessible*, and when some marked state is reachable from any state it is said to be *coaccessible*. A process which is both accesible and coaccessible is called *trim*, and a trim transition machine can thus reach all of its states, and from every state some marked state can be reached; the process is live.

**Definition 2.15 Accessible**

For a transition machine $P$ a state $q \in Q_P$ is said to be *accessible*, if $q$ can be reached from some initial state. That is

$$q \in Q_P \text{ is accessible} \Leftrightarrow \exists t_P \in tr(P) \text{ such that } last(t_P) = q. \tag{2.18}$$

When all states of $Q_P$ are reachable we will say that $P$ is accessible. That is

$$P \text{ is accessible} \Leftrightarrow \forall q \in Q_P \ \exists t_P \in tr(P) \text{ such that } last(t_P) = q. \tag{2.19}$$

**Remark.** Some authors make a global assumption that all transition machines are accessible, see for instance Eilenberg (1974).. There is little gained by allowing non-accessible states. However, in this work we will generally not make this assumption. Only in very specific places will we assume the transition machines to be accessible, and then this fact will be mentioned explicitly. At other times, it turns out that some of the proofs that are to be given become simpler when we do not have to consider whether a specific transition machine is accessible or not. □

Some states of a transition machine have the property that there exists at least one trace of the closed traceset reaching that state. For instance, the empty trace, $\varepsilon$, is considered to reach all the states of $I_P$. Thus, when a transition machine is accessible, such atates are the only states. This results in the following lemma.

**Lemma 2.16** When a transition machine $P$ is accessible, its traceset defines its state-space. That is,

$$P \text{ accessible} \Leftrightarrow Q_P = last(tr(P)). \tag{2.20}$$

**Remark.** Thus, for an accessible transition machine every edge is a subtrace (of length 1) of some trace of that machine, and the union over all subtraces of length 1 of all traces is equal to the edge set. That is, when $P$ is accessible $E_P = \bigcup_{tet' \in tr(P)} e$. This means that for an accessible transition machine, the traceset holds all information conveyed by the state-space and the edge-set. We can also note that when $P$ is accessible, any $t \in E_P^*$ is a subtrace of some $t_P \in tr(P)$. □

**Definition 2.17 Coaccessible**

For a transition machine $P$ a state $q \in Q_P$ is said to be *coaccessible* if from $q$ *some* marked state can be reached. That is

$$q \in Q_P \text{ coaccessible} \Leftrightarrow \exists t \in E_P^* \ first(t) = q \wedge last(t) \in M_P. \tag{2.21}$$

When all states of $Q_P$ are coaccessible, we say that $P$ is coaccessible. That is

$$P \text{ coaccessible} \Leftrightarrow \forall q \in Q_P \ \exists t \in E_P^* \ first(t) = q \wedge last(t) \in M_P. \tag{2.22}$$

**Remark.** Note that any state reaches itself by the null trace, thus any marked state is naturally coaccessible. Note also that when $M_P = \emptyset$ but $Q_P \neq \emptyset$ the process is inherently non-coaccessible. Therefore, we will always assume that for any given process $P$ with $Q_P \neq \emptyset$, $M_P$ is also nonempty. This has the, seemingly unfortunate, consequence of requiring a "non-marked" process to have *all* its states marked, that is $M_P = Q_P$. However, there is a lot gained by this assumption, and it seems unavoidable unless we refrain totally from mixing marked and unmarked automata. □

**Definition 2.18 Trim**

A transition machine $P$ is said to be *trim* if it is both accessible and coaccessible. That is

$$P \text{ trim} \Leftrightarrow \forall q \in Q_P \; \exists tt' \in tr_m(P) \; last(t) = q \tag{2.23}$$

**Remark.** This definition means that for $P$ to be trim, every state must be reached by some trace beginning at an initial state and ending in a marked state.

The fact that a transition machine is trim if it is both accessible and coaccessible, suggests an effective method of finding the trim part of a given transition machine. This will be of importance in Chapter 4 where we attempt to find a supervisor such that the supervised system is trim. □

**Lemma 2.19** A transition machine $P$ is trim if the prefix closure of its marked traceset is equal to its closed traceset. That is,

$$P \text{ trim} \Rightarrow \overline{tr_m(P)} = tr(P). \tag{2.24}$$

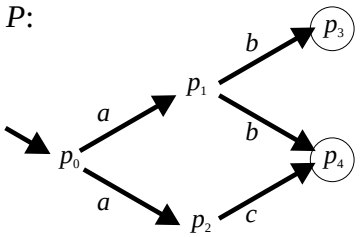
**Proof.** We only show that when a transition machine is trim, then its closed traceset is a subset of the prefix closure of its marked traceset, that is, $P$ trim $\Rightarrow tr(P) \subseteq \overline{tr_m(P)}$. Together with the fact that for any transition machine $\overline{tr_m(\cdot)} \subseteq tr(\cdot)$, it then follows that whenever $P$ is trim $\overline{tr_m(P)} = tr(P)$.

($\Rightarrow$) When $P$ is trim it is also accessible, so that for all states $q \in Q_P$ there exists a trace $t \in tr(P)$ such that $q = last(t)$. Furthermore, $P$ is also coaccessible, so that there exists a trace $t' \in E_P^*$ such that $t < t'$ and $last(t') \in M_P$. Thus $tt' \in tr_m(P)$ and $t \in \overline{tr_m(P)}$.

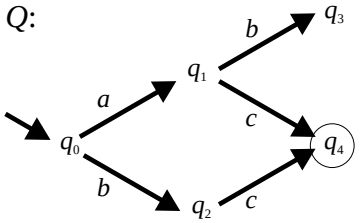
■

**Remark.** Since a trim transition machine is both accessible and coaccessible, all states can be reached from some initial state, and all states can reach some marked state. Therefore, every state is "passed" by some trace on the way to a marked state, and the prefix closure of $tr_m(\cdot)$ reaches all states. This is why $\overline{tr_m(\cdot)} = tr(\cdot)$.

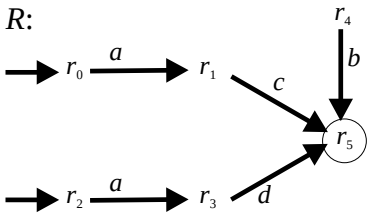
Note though, that the converse does not necessarily hold. The prefix closure of the marked traceset can be equal to the closed traceset, without the transition machine being trim. This is so, since when $\overline{tr_m(\cdot)} = tr(\cdot)$, all accessible states, given by $tr(\cdot)$, are also coaccessible, given by $\overline{tr_m(\cdot)}$; non-accessible states are not concerned. However, when a transition machine is trim there does not exist any non-accessible states. □



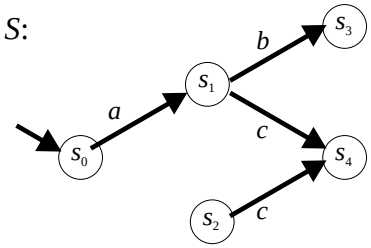

$$Q_P = \{p_0, p_1, p_2, p_3, p_4\}$$
$$\Sigma_P = \{a, b, c\}$$
$$I_P = \{p_0\}$$
$$M_P = \{p_3, p_4\}$$
$$E_P = \{(p_0, a, p_1), (p_0, a, p_2), (p_1, b, p_3), (p_1, b, p_4), (p_2, c, p_4)\}$$



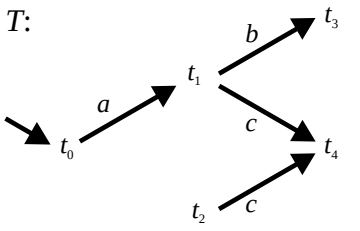

$$Q_Q = \{q_0, q_1, q_2, q_3, q_4\}$$
$$\Sigma_Q = \{a, b, c, d\}$$
$$I_Q = \{q_0\}$$
$$M_Q = \{q_4\}$$
$$E_Q = \{(q_0, a, q_1), (q_0, b, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, c, q_4)\}$$




$$Q_R = \{r_0, r_1, r_2, r_3, r_4, r_5\}$$
$$\Sigma_R = \{a, b, c, d\}$$
$$I_R = \{r_0, r_2\}$$
$$M_R = \{r_5\}$$
$$E_R = \{(r_0, a, r_1), (r_2, a, r_3), (r_1, c, r_5), (r_3, d, r_5), (r_4, b, r_5)\}$$




$$Q_S = \{s_0, s_1, s_2, s_3, s_4\}$$
$$\Sigma_S = \{a, b, c\}$$
$$I_S = \{s_0\}$$
$$M_S = Q_S$$
$$E_S = \{(s_0, a, s_1), (s_1, b, s_3), (s_1, c, s_4), (s_2, c, s_4)\}$$




$$Q_T = \{t_0, t_1, t_2, t_3, t_4\}$$
$$\Sigma_T = \{a, b, c\}$$
$$I_T = \{t_0\}$$
$$M_T = \varnothing$$
$$E_T = \{(t_0, a, t_1), (t_1, b, t_3), (t_1, b, t_4), (t_2, c, t_4)\}$$

**Figure 2.1:** The transition machine $P$ is both accessible and coaccessible, and therefore trim. Since from $q_3$ no marked state can be reached, $Q$ is not coaccessible. The machine $R$ is not accessible since $r_4$ cannot be reached from any of the two initial states. For $S$ all states are marked, while for $T$ no states are marked, therefore, $S$ is coaccessible but $T$ is not. Neither of $S$ nor $T$ is accessible. Note also that $P$ and $R$ are nondeterministic, since from the state $p_0$ the $a$ event leads either to $p_1$ or $p_2$, while $R$ has two initial states. Furthermore, the alphabet $\Sigma_Q$ contains the event $d$, which is not present in any edge of $E_Q$.

**Summary**

In this section we have defined the behavior of a finite transition machine. This behavior is described by traces that record the states and events that the process visits and executes due to its transiting between states. Some states are distinguished to be of specific importance, so that a subset of the possible traces is regarded as significant; that is, they constitute the marked traceset.

The notion of a trim transition machine will be very important in the following sections. The goal is to control a transition machine so that it can always reach some marked state. Lemma 2.19 shows that trimness is sufficient to be able to reach all marked states from the set of initial states.

See Figure 2.1 for a number of transition machines that aim to illustrate the notions of accessibility, coaccessibility and trimness, described in the latter part of this section.

### 2.2.3 Strings and Languages

The label of an edge is the event that either occurs as a consequence of the transition from $first(\cdot)$ to $last(\cdot)$ of that edge, or triggers that transition. In the former case we say that the event is *generated* by the transition, and in the latter case the transition, or rather the transition machine, *follows* the event. We will for the most part not distinguish between event generation and event following, (formally there is no difference), and we will merely say that the event or transition is *executed*. As the transition machine evolves, it transits between states, executing events. The sequences of events are called *strings*, and are given by the labels of the traces. Thus, many of the definitions and proofs concerning transitions have corresponding definitions and proofs regarding the strings. Note, however, that in general the strings that are executed hold less information than the traces, since traces can differ in the visited states, while having the same label. Thus, one and the same string cannot distinguish between similar traces.

**Definition 2.20 Substring**

A *substring* of a string $\varsigma \in \Sigma^*$ is any string $s \in \Sigma^*$ such that

$$\exists s', s'' \in \Sigma^* \text{ such that } \varsigma = s'ss''. \tag{2.25}$$

**Remark.** The definitions of *prefix* and *suffix* of a string correspond to the definitions given for traces in Definition 2.9. □

**Definition 2.21 Closed and Marked Language**

The set of event strings represented by a transition machine $P$ is called its *closed language*, and is defined as

$$L(P) = \{s \in \Sigma_P^* \,|\delta(P, s) \neq \emptyset\}. \tag{2.26}$$

The subset of the closed language reaching marked states is called the *marked language*, and is defined as

$$L_m(P) = \{s \in L(P) \,|\delta(P, s) \bigcap M_P \neq \emptyset\}. \tag{2.27}$$

**Remark.** The closed traceset and the closed language of a transition machine $P$ are related as $L(P) = label(tr(P)) = \{s \in \Sigma_P^* \,|\exists t \in tr(P) \;\text{ such that } s = label(t)\}$. Thus, the empty string $\varepsilon \in L(P)$.

The marked traceset and the marked language are related as $L_m(P) = label(tr_m(P)) = \{s \in \Sigma_P^* \,|\exists t \in tr_m(P)\; s = label(t)\}$. A string may, in general, lead to a number of states. If one of these is marked, then that string belongs to the marked language. □

**Definition 2.22 Nonblocking**

For a transition machine $P$ with marked and closed languages $L_m(P)$ and $L(P)$, respectively, $L(P)$ is said to be *nonblocking* if every string of $L(P)$ can be continued into a string of $L_m(P)$. That is,

$$L(P) \text{ nonblocking} \Leftrightarrow \overline{L_m(P)} = L(P). \tag{2.28}$$

**Remark.** A transition machine with a nonblocking language is such that for every string there exists some accessible state that is also coaccessible. That is

$$\overline{L_m(P)} = L(P) \Leftrightarrow \forall s \in L(P)\; \exists p \in \delta(P,s)\; \exists s' \in \Sigma_P^*\; \delta_P(p,s') \bigcap M_P \neq \emptyset. \tag{2.29}$$

From the definitions of the marked and closed tracesets and languages, these implications follow,

$$P \text{ trim} \Rightarrow \overline{tr_m(P)} = tr(P) \Rightarrow \overline{L_m(P)} = L(P). \tag{2.30}$$

That $P$ trim only implies that $\overline{tr_m(P)} = tr(P)$ was shown by Lemma 2.19. That this, in turn, only implies that $P$ is nonblocking is a consequence of the fact that different traces can have the same label. Thus, $P$ can be nonblocking without the prefix closure of the marked traceset being equal to the closed traceset. □

The way we have defined a transition machine, the events executable in a specific state may be only a subset of the events defined by the transition machines alphabet. The set of events that are defined by edges with that state as $first(\cdot)$ is called the *ready set* of that state.

**Definition 2.23 Ready Set**

The *ready set* of a state $p \in Q_P$ is the set of events leading out of that state, defined as

$$\begin{aligned} \text{out}(p) &= \{\sigma \in \Sigma_P \,|\exists e \in E_P \; first(e) = p \wedge label(e) = \sigma\} \\ &= \{\sigma \in \Sigma_P \,|\delta_P(p,\sigma) \neq \emptyset\} \end{aligned} \tag{2.31}$$

**Remark.** A transition machine (or state machine) with the ready set of some state being a subset of the alphabet is sometimes said to be *incompletely* specified (or *incomplete*), see Eilenberg (1974), or even *nondeterministic*, see Hopcroft (1979). It can be shown, see Eilenberg (1974), that an incompletely specified state machine can always be converted to a completely specified state machine by adding a non-marked "junk" state. All states that do not have their ready sets equal to the entire alphabet are then augmented with the missing events, transiting to the junk state. This does not alter the *marked language* represented by that machine, albeit, the closed language would be altered. We feel that our

definition is an intuitive one; most discrete event processes cannot execute all events in all states. Furthermore, requiring the transition machines to be completely specified removes the ability for one transition machine to control another. Therefore, we do not include this requirement. Note also, that we reserve the term *nondeterministic* to mean something else. An incompletely specified transition machine is not necessarily nondeterministic, see Definition 2.28. □

Naturally, the ready set of a state is allowed to be empty. Thus, no further execution of events can occur from that state, and the execution of the transition machine can be seen to have terminated. Thus, such states are called terminating states.

**Definition 2.24 Terminating State**

A state $q \in Q^+$ with an empty ready set will be called a *terminating* state. That is,

$$q \text{ terminating} \Leftrightarrow \text{out}(q) = \emptyset. \tag{2.32}$$

**Remark.** A terminating state is not necessarily non-coaccessible. If $q$ is marked, then from $q$ a marked state can always be reached, namely $q$ itself. Thus, $q$ is then by definition coaccessible, even though it may be terminating. □

A completely specified transition machine $P$ will, after any string, have the entire alphabet as possible continuation. Hence its closed language is $L(P) = \Sigma_P^*$. Not so, however, when the transition machine is allowed to be incompletly specified. After a string $s$ only a subset of the alphabet is possible as continuation of $s$ in $P$. This subset is called the *active set* of $L(P)$ after $s$.

**Definition 2.25 Active Set**

For a string $s \in L(P)$ the *active set*, $\gamma(L(P), s)$, is the set of events that can follow $s$ in the language $L(P)$. That is

$$\begin{aligned} \gamma(L(P), s) &= \{\sigma \in \Sigma_P \,|s\sigma \in L(P)\} \\ &= \bigcup_{p \in \delta(P,s)} \text{out}(p) \end{aligned} \tag{2.33}$$

**Remark.** Note that, the active set of a string is the union of the ready sets of the states reachable by that string. This is an important fact that will be used to examine properties of nondeterministic supervisors, such as completeness and controllability in Section 3.4. The difference between the ready sets and the active set after a string $s$ is depicted in Figure 2.2.

Sometimes we will speak of the "active events after $s$", and mean the active set of $s$. We will use these two expressions interchangeably. □

The following lemma will be used when showing how structural properties of automata and their languages relate. This is important in characterizing transition machines able to control other transition machines within some prespecified specification.

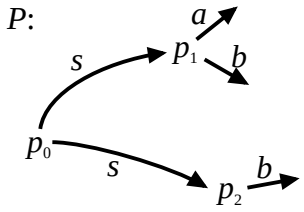



**Figure 2.2:** The active set after the string $s$ is $\gamma(L(P), s) = \{a, b\}$. This is the same as the ready set of the state $p_1$. However, for the state $p_2$, the ready set is $\text{out}(p_2) = \{b\}$.

**Lemma 2.26** For two transition machines $S$ and $P$, when the language of $S$ is a subset of the language of $P$, then, and only then, the active set of any string of $S$ is a subset of the active set of the same string of $P$. That is,

$$L(S) \subseteq L(P) \Leftrightarrow \forall s \in L(S)\ \gamma(L(S), s) \subseteq \gamma(L(P), s). \tag{2.34}$$

**Proof.** ($\Rightarrow$) Assume that $L(S) \subseteq L(P)$ but there existst a string $s \in L(S)$ with $\gamma(L(S), s) \not\subseteq \gamma(L(P), s)$. Then, there exists an event $\sigma \in \Sigma_P$ such that $s\sigma \in L(S)$ but $s\sigma \notin L(P)$. Thus, $L(S)$ cannot be a subset of $L(P)$.

($\Leftarrow$) Assume that for all strings $s \in L(S)$, $\gamma(L(S), s) \subseteq \gamma(L(P), s)$ but $L(S) \not\subseteq L(P)$. Then there exists a string $t\sigma \in L(S)$ such that $t\sigma \notin L(P)$. Of course, then $\sigma \in \gamma(L(S), t)$ and $\sigma \notin \gamma(L(P), t)$, so that $\gamma(L(S), t) \not\subseteq \gamma(L(P), t)$.

These two contradictions prove Lemma 2.26. ■

**Remark.** By Lemma A.6 the right hand side of (2.34) is equivalent to $\forall s \in L(S)\ \forall q \in \delta(S, s)\ \text{out}(q) \subseteq \gamma(L(P), s)$. This means that after a string present in both languages, the ready sets of all the states of $S$ reached by that string are subsets of the active set of $L(P)$ after that string. Note that this, as well as Lemma 2.26, holds regardless of whether the two transition machines have mutual alphabets or not. However, if $\Sigma_P \subseteq \Sigma_S$ then it is obvious that the ready sets of the states of $Q_S$ cannot include the events $\Sigma_S - \Sigma_P$, otherwise $L(S)$ cannot be a subset of $L(P)$. □

By Lemma A.2 of Appendix A, Lemma 2.26 immediately leads to the following corollary, which will also be needed later.

**Corollary 2.27** For two transitions machines $S$ and $P$ and any subset $A \subseteq \Sigma_P$ we have that

$$L(S) \subseteq L(P) \Rightarrow \forall s \in L(S)\ \gamma(L(S), s) \bigcap A \subseteq \gamma(L(P), s) \bigcap A. \tag{2.35}$$

Structural properties of certain transition machines make them behave "nicely". For instance, when a transition machine is *deterministic* its traces are uniquely defined by its strings. By deterministic we mean the following.

**Definition 2.28 Deterministic Transition Machine**

A transition machine $P$ will be called *deterministic*, if and only if the set of initial states contains exactly one element and every edge is uniquely defined by its $first(\cdot)$ and $label(\cdot)$. That is

$$P \text{ deterministic} \Leftrightarrow \begin{cases} |I_P| = 1 \\ \forall e, e' \in E_P \; first(e) = first(e') \wedge label(e) = label(e') \rightarrow e = e' \end{cases} \tag{2.36}$$

**Remark.** The second requirement can also be expressed as $\forall q \in Q_P \; \forall s \in \Sigma_P^* \; |\delta_P(q, s)| \leq 1$. This has the consequence that for a deterministic process every string reaches one, and only one, state. This has the immediate consequence that a transition machine $P$ is deterministic if and only if all similar traces of its traceset are also equal. Thus, every state is reachable by unique strings reaching only that state, see Hopcroft (1979). □

Thus, a transition machine is called *deterministic* if and only if

1. The set of initial states contains (at most) one element, and
2. Every string uniquely defines one and only one reached state.

Consequently, a transition machine is *nondeterministic* if

1. The set of initial states contains more than one element, or
2. There exists at least one string reaching more than one state.

For generality as well as for ease of notation we will formally consider transition machines with multiple initial states. However, with a few exceptions, we will only show transition machines with a single initial state. This is in keeping with the tradition of Eilenberg (1974), where only automata with single initial states are shown as examples, but multiple initial states are formally considered. We can note also that Hopcroft (1979) does not consider multiple initial states at all.

### Summary

In this section we have only considered the event sequences executed by a transition machine. Thus, no information of the states visited during the event execution is included in such a model. It is important to note that only when a transition machine is deterministic, can it be fully described by its event sequences, see the remark to Definition 2.28. The supervisory control theory of Ramadge (1987) and Wonham (1987) originally considered only deterministic automata so that a language representation was sufficient. However, when generalizing to nondeterministic transition machines this is no longer so, as will be shown in Chapter 3.

Of specific importance in this section are the definitions of the active set, Definition 2.25, and the ready set, Definition 2.23. Note that the active set defined by a given string is the *union* of the ready sets of the states reached by that string. It is precisely this fact that complicates matters when considering nondeterministic processes. Of importance are also the properties of sublanguages given by Lemma 2.26 and Corollary 2.27. These will be used when considering *uncontrollable* events and contribute to conditions under which a transition machine can in fact be controlled by another transition machine to exhibit a specified behavior.

## 2.3 Refinement And Subprocesses

Some processes have structural properties relating them to each other. These structural properties are important for the correct application of the algorithms to be described. Therefore, two ordering relationships among processes will be defined, *subprocess* and *refinement*. Note that we will only consider closed tracesets in this section, including the marked states without much notice. However, since the marked traceset is included in the closed traceset, this will constitute no problem.

### 2.3.1 Subprocesses

When the structural appearance of a process $S$ is contained within another process $P$, $S$ is said to be a *subprocess* of $P$. The process $P$ is termed the *superprocess* of $S$. An intuitive definition of a subprocess as having subsets of the state-space and the edge-set of its superprocess will be given. However, another definition of a subprocess will also be given, not requiring the state-space of the subprocess to be a subset of the state-space of the superprocess. This latter definition may seem to be less constricting of the superprocess, but it will be shown that the two definitions are in fact eqivalent, in the sense that the number of *different* subprocesses that they define is the same.

We begin with the intuitive definition of a subprocess.

**Definition 2.29 Subprocess**

For two transition machines $S = (Q_S, \Sigma_S, I_S, M_S, E_S)$ and $P = (Q_P, \Sigma_P, I_P, M_P, E_P)$, with $\Sigma_S = \Sigma_P$, $S$ is said to be a *subprocess* of $P$ if its state-space is included in $P$ 's, and $S$ has smaller or equal structural behavior at each state. That is,

$$S \leq P \Leftrightarrow \begin{cases} Q_S \subseteq Q_P \\ \Sigma_S = \Sigma_P \\ I_S = I_P \bigcap Q_S \\ M_S = M_P \bigcap Q_S \\ E_S \subseteq E_P \end{cases} \tag{2.37}$$

**Remark.** Note that, while $Q_S$ and $E_S$ are allowed to be subsets of $Q_P$ and $E_P$, $\Sigma_S$ is required to be equal to $\Sigma_P$. Thus, in forming a subprocess of a given process, we are allowed to remove states and edges, but not to remove events. Furthermore, if an initial or marked state of $P$ is in the state-space of $S$, then this state must also be an initial or marked state, respectively, of $S$.

□

In Figure 2.3 the $S_i$ are all subprocesses of their respective $P_i$ processes. However, $S_2$ is not a subprocess of $P_3$, even though $Q_{S_2} \subseteq Q_{P_3}$, since an edge $(p_1, b, p_2)$ belongs to $E_{S_1}$ but not to $E_{P_2}$. Note also that $S_2$ is a subprocess of both $P_2$ and $P_1$, as well as $S_1$. Furthermore, the state $p_3$ is non-accessible in $S_1$, but not in $P_1$. Removing this state from $Q_{S_1}$ would not alter the closed traceset of $S_1$, $tr(S_1)$ would still be a subset of $tr(P_1)$, but $S_1$ would no longer be a subprocess of $P_1$. Finally, note that $P_1$ is nondeterministic due to the edges $(p_0, a, p_1)$ and $(p_0, a, p_3)$ both belonging to $E_{P_1}$.

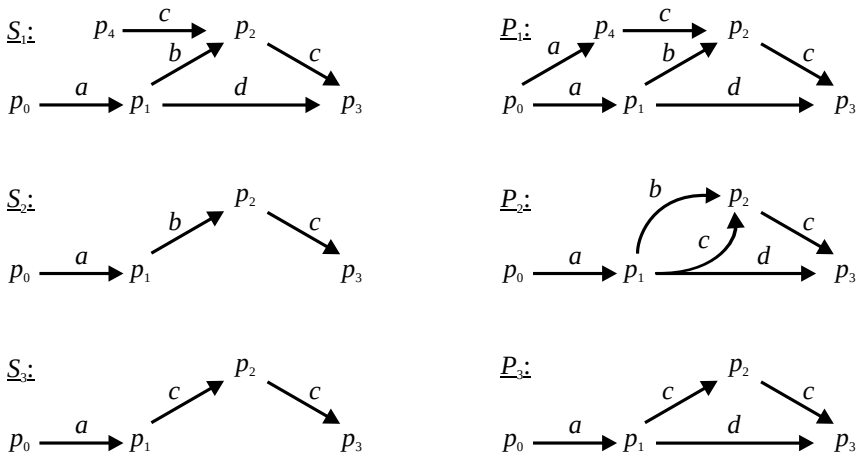



**Figure 2.3:** Examples of subprocesses and superprocesses. $S_1$ is a subprocess of $P_1$. $S_2$ is a subprocess of both $P_1$ and $P_2$, while $S_3$ is a subprocess of both $P_2$ and $P_3$. Note also that $P_3 \leq P_2$ and that $P_1$ is nondeterministic.

The subprocess relation defines a partial ordering on transitions machines. This is shown by the following lemmas and theorem.

**Lemma 2.30** The subprocess relation is *reflexive*, that is, any transition machine is a subprocess of itself.

**Proof.** This is immediate by Definition 2.29. ∎

**Lemma 2.31** The subprocess relation is *transitive*, that is for three transition machines, $S$, $P$ and $Q$, if $S \leq P$ and $P \leq Q$ then $S \leq Q$.

**Proof.** Again this is obvious by Definition 2.29. When $S \leq P$ and $P \leq Q$ we have that $Q_S \subseteq Q_P \subseteq Q_Q$, and $E_S \subseteq E_P \subseteq E_Q$, Also, $I_S = I_P \bigcap Q_S$ and $I_P = I_Q \bigcap Q_P$ so that $I_S = I_Q \bigcap Q_P \bigcap Q_S = I_Q \bigcap Q_S$, and similarily for $M_S$. Thus, $S \leq Q$. ∎

**Lemma 2.32** The subprocess relation is *antisymmetric*, that is for two transition machines, $S$ and $P$, each is a subprocess of the other if and only if they are equal. That is,

$$S \leq P \wedge P \leq S \Leftrightarrow S = P \tag{2.38}$$

**Proof.** By Definition 2.29 we have that

$$S \leq P \wedge P \leq S \Leftrightarrow \left\{ \begin{array}{c} Q_S \subseteq Q_P \wedge Q_P \subseteq Q_S \\ \Sigma_S = \Sigma_P \\ I_S \subseteq I_P \wedge I_P \subseteq I_S \\ M_S \subseteq M_P \wedge M_P \subseteq M_S \\ E_S \subseteq E_P \wedge E_S \subseteq E_P \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{c} Q_S = Q_P \\ \Sigma_S = \Sigma_P \\ I_S = I_P \\ M_S = M_P \\ E_S = E_P \end{array} \right\} \Leftrightarrow S = P \tag{2.39}$$

■

**Remark.** This defines what we regard as equivalent processes, each is a subprocess of the other. This is certainly a strong notion of equivalence of processes. Other definitions of equivalence have been proposed. A discussion of these and other topics related to different approaches for modeling finite state-machines is given in Section 2.5. □

A binary relation is said to be a *partial ordering* if it is reflexive, transitive and antisymmetric. From Lemmas 2.30, 2.31 and 2.32 we know that the subprocess relation is reflexive, transitive and antisymmetric. This leads to the following theorem.

**Theorem 2.33** The subprocess relation defines a *partial ordering* on transition machines.

The partial ordering of the subprocess relation can also be seen in Figure 2.3. For instance $S_2 \leq S_1 \leq P_1$. Note also that, even though $S_2$ and $S_3$ are both subprocesses of $P_2$, they are not related to each other. Neither is a subprocess of the other; they are *incomparable*. This comes, of course, from the fact that some edge of $E_{S_2}$ does not belong to $E_{S_3}$, and vice versa.

Since both the state-space and the event set of any process is finite, so is the edge-set. Thus, the number of subprocesses that can be obtained by removing states and transitions is also finite. Therefore the set of all subprocesses of a given process can be readily defined.

**Definition 2.34 Set of All Subprocesses**

For a given transition machine $P$, the *set of all subprocesses* is defined as

$$\mathcal{S}(P) = \{S \in \mathcal{P} \,|\, S \leq P\} \tag{2.40}$$

where $\mathcal{P}$ denotes the universe of discourse as defined by Definition 2.4.

There are a number of useful properties regarding subprocesses in relation to their superprocesses. For instance, when $S$ is a subprocess of $P$, the traceset of $S$ is a subset of the traceset of $P$. That is $S \leq P \Rightarrow tr(S) \subseteq tr(P)$. When $S$ and $P$ are accessible and their alphabets are equal, the converse also holds. This is obvious since the edge-set of $S$ is a subset of the edge-set of $P$, and the initial states of $S$ are also initial states for $P$. This also means that the language of $S$ is a subset of the language of $P$. Naturally, both of these properties hold equally for the marked tracesets and languages. Note though that a subprocess is not necessarily marked, that is $M_S = M_P \bigcap Q_S$ may be empty. However, by Definition 2.17, such a subprocess is not coaccessible (unless $Q_S$ is also empty) and thus $tr_m(\cdot) = \emptyset$ which is, of course, a subset of any marked trace set.

**Lemma 2.35** For a subprocess $S$ of a superprocess $P$, we have that $tr(S) \subseteq tr_m(P) \Leftrightarrow tr_m(S) = tr(S)$.

**Proof.** ($\Rightarrow$) Since $tr(S)$ is closed and $tr(S) \subseteq tr_m(P)$, every state reached by $tr(S)$ must be marked. That is, it must hold that $last(tr_m(S)) = last(tr(S))$, and thus $tr_m(S) = tr(S)$.

($\Leftarrow$) When $tr_m(S) = tr(S)$, $last(tr_m(S)) = last(tr(S)) \subseteq M_P$ and therefore $tr(S) \subseteq tr_m(P)$. ■

Since $tr(S)$ is closed, so is $tr_m(S)$ when they are equal. This means that the prefix closure of the marked traceset is equal to the marked traceset, as well as to the closed traceset. Therefore, Lemma 2.35 has the corollary that whenever the closed traceset of the subprocess is a subset of the marked traceset of the superprocess, the closed traceset of the subprocess is equal to the prefix closure of its marked traceset.

**Corollary 2.36** For a subprocess $S$ of a superprocess $P$, we have that $tr(S) \subseteq tr_m(P) \Leftrightarrow \overline{tr_m(S)} = tr(S)$.

**Remark.** Thus, when $S$ is accessible, $tr(S) \subseteq tr_m(P)$ means that $S$ is trim. See Definition 2.18. □

We also have a lemma relating the definition of subprocesses as given in Definition 2.29 to the transition function as defined in Definition 2.5.

**Lemma 2.37** For a subprocess $S$ of a process $P$, for all states of $Q_S$ and for all events of $\Sigma_S$, the value of the transition function of $S$ is a subset of the value of the transition function of $P$. That is,

$$S \leq P \Rightarrow \forall q \in Q_S \ \forall \sigma \in \Sigma_S \ \delta_S(q, \sigma) \subseteq \delta_P(q, \sigma). \tag{2.41}$$

**Proof.** This comes from the fact that when $S \leq P$ then $Q_S \subseteq Q_P$ and $E_S \subseteq E_P$.

($\Rightarrow$) Assume that $E_S \subseteq E_P$ but that there exists a state $q \in Q_S$ and an event $\sigma \in \Sigma_S$ such that $\delta_S(q, \sigma) \not\subseteq \delta_P(q, \sigma)$. Then there exists a state $q' \in \delta_S(q, \sigma)$ such that $q' \notin \delta_P(q, \sigma)$. By the remark to Definition 2.5, this is equal to $(q, \sigma, q') \in E_S$ and $(q, \sigma, q') \notin E_P$. This clearly contradicts the initial assumption.

■

**Remark.** This merely means to state the obvious fact that when $E_S$ is a subset of $E_P$ then the transition functions are as above. However, we can say nothin about states not included in the edge-sets. When the processes are accessible, though, the converse of Lemma 2.37 also holds. □

We can note that the intuitive definition of a subprocess places an unnecessary constraint on the state-space of the subprocess. In essence Definition 2.29 says that the *structure* of a subprocess is contained within its superprocess. Since the structure of a process is independent of the naming of its state-space, we can relax the constraint that the state-space of the subprocess has to be included in the state-space of the superprocess. This can be done by defining a subprocess through the existence of an injective state-mapping between the subprocess' state-space and the state-space of its superprocess.

**Definition 2.38 Subprocess**

For two transition machines $S = (Q_S, \Sigma_S, I_S, M_S, E_S)$ and $P = (Q_P, \Sigma_P, I_P, M_P, E_P)$, with $\Sigma_S = \Sigma_P$, $S$ is said to be a *subprocess* of $P$ if there exists an *injective* (total) function $f: Q_S \to Q_P$, such that for all traces $t_S \in E_S^*$ there exists a trace $t_P \in E_P^*$ such that

$$\begin{aligned} f(first(t_S)) &= first(t_P) \\ label(t_S) &= label(t_P) \\ f(last(t_S)) &= last(t_P) \end{aligned} \tag{2.42}$$

and

$$\forall q \in Q_S \begin{cases} q \in I_S \Leftrightarrow f(q) \in I_P \\ q \in M_S \Leftrightarrow f(q) \in M_P \end{cases} \tag{2.43}$$

**Remark.** The function $f$ being injective means that it is one-to-one. That is, distinct states of $Q_S$ are mapped onto distinct states of $Q_P$. Obviously, $f$ can be injective only if the number of states in $Q_S$ is less than or equal to the number of states in $Q_P$. Since, by definition, every state has at least the null trace, $\varepsilon$, *all* states of $Q_S$ are mapped into $Q_P$ by $f$. That is, the *domain* of $f$ is $D_f = Q_S$. Thus, nonaccessible states are also covered by this definition. Note though, that the *range* of $f$ is $R_f \subseteq Q_P$, not necessarily the entire state-space $Q_P$. If both $D_f$ and $R_f$ are the entire state-spaces $Q_S$ and $Q_P$, respectively, then $f$ is said to be *bijective*. This means that every distinct state of $Q_S$ is mapped onto a distinct state of $Q_P$, and every state of $Q_P$ has a corresponding state in $Q_S$. Of course, when the $f$ is bijective, the number of states of $Q_S$ is equal to the number of states of $Q_P$; in fact, the two processes are equal.

Note that, when $P$ is nondeterministic there may exist many such functions $f$ for one and the same subprocess $S$ of $P$, all of which satisfy Definition 2.38. However, the existence of *any* such function from $Q_S$ into $Q_P$ is enough to make $S \leq P$. □

Definition 2.38 has the consequence that for all pairs of related traces of the subprocess, there must exist pairs of respectively similar traces of the superprocess that are also related. Furthermore, for every pair of diverging traces of the subprocess, there must exist pairs of respectively similar diverging traces of the superprocess. In Figure 2.5 on page 43 all the $S_i$ are subprocesses of the respective $P_i \parallel S_i$, as can be verified by inspection. However, none of the $P_i$ are subprocesses of $P_i \parallel S_i$, since in every $P_i$ the strings $t$ and $u$ are related, but these strings are not related in $P_i \parallel S_i$. Also $S_4$ is a subprocess of $P_4$ since there exist similar traces that are related as well as unrelated in both transition machines. Note that we can establish this without knowledge of the state-space of the processes. What we do is that we determine the function defined in Definition 2.38.

Definition 2.38 concerns the traces defined by the sets $E_S^*$ and $E_P^*$. This so, since we want the definition of a subprocess to encompass nonaccessible states; a subprocess is not allowed to have nonaccessible states that have no corresponding state in the state-space of the superprocess. However, since $E_S \subseteq E_S^*$, and $E_P \subseteq E_P^*$ individual edges of $E_S$ are mapped onto distinct edges of $E_P$. Edges are essentially traces of length 1, and Definition 2.38 places no restriction on the length of the traces; they can be of length 0, 1 or $n$ (for finite $n$, of course).

The two definitions of subprocesses, Definition 2.29 and Definition 2.38, may seem vastly different. However, it can be shown that they are in fact equivalent in the sense that the set of different subprocesses of a given process defined by each definition can

be mapped onto each other by bijective mappings. Thus, for every element defined by Definition 2.29 there exists an element in Definition 2.38 such that the two elements are considered equal in the sense that each is a subprocess of the other.

**Lemma 2.39** Definition 2.29 is equivalent to Definition 2.38.

**Proof.** To prove this, let us denote by $S_1(P)$ the set of all subprocesses of $P$ as defined by Definition 2.29, and by $S_2(P)$ the set of all subprocesses defined by Definition 2.38. Let us also denote by $f(Q_S)$, the subset of $Q_P$ to which the injective function $f$ maps the states of $Q_S$.

($\Rightarrow$) It is obvious that for any element $S_1 \in S_1(P)$ there does exist an injective mapping function such as given by Definition 2.38, namely the function $f_1: Q_{S_1} \to Q_P$ such that $\forall p \in Q_{S_1}\ f_1(p) = p \in Q_P$. This so, since $Q_{S_1} \subseteq Q_P$. Thus, the set of subprocesses defined by Definition 2.29 is a subset of the set defined by Definition 2.38, so that the implication holds.

We can note that, since $f_1$ maps $Q_{S_1}$ onto the subset of $Q_P$ that is $Q_{S_1}$ itself, $f_1$ is in fact the identity function, $I_1$, on $Q_{S_1}$. Thus, the inverse $f_1^{-1}$ is also equal to $I_1$.

($\Leftarrow$) Assume that for $S_1 \in S_1(P)$ and $S_2 \in S_2(P)$ there exists the injective functions $f_1: Q_{S_1} \to Q_P$ and $f_2: Q_{S_2} \to Q_P$, respectively. $f_1$ is $I_1$ as defined above, with $D_{f_1} = Q_{S_1}$ and $R_{f_1} = Q_{S_1} \subseteq Q_P$, while $f_2$ exists by Definition 2.38 and $D_{f_2} = Q_{S_2}$ and $R_{f_2} \subseteq Q_P$. We have to show that whenever $f_1(Q_{S_1}) = f_2(Q_{S_2})$ then there exists a bijective function $g$, such that $g(Q_{S_1}) = Q_{S_2}$ and $g^{-1}(Q_{S_2}) = Q_{S_1}$. If so, then $S_1$ and $S_2$ are equal in the sense that they are subprocesses, as defined by Definition 2.38, of each other.

For $f_1$, its inverse exists and is $I_1$, with both domain and range $Q_{S_1}$. Since $f_2$ is injective, the inverse $f_2^{-1}$ exists. Also, $D_{f_2^{-1}} = R_{f_2}$ and $R_{f_2^{-1}} = D_{f_2}$. However, since $R_{f_2} \subseteq Q_P$, $f_2^{-1}$ is *not* a function from $Q_P$ to $Q_{S_2}$, but merely $f_2^{-1}: R_{f_2} \to Q_{S_2}$. Define $g = I_1 \circ f_2$, where $\circ$ denotes function composition. The composition of two injective functions is also injective. We have that $D_g = D_{f_2} = Q_{S_2}$ and $R_g = D_{I_1} = Q_{S_1}$. The inverse function $g^{-1}$ is then equal to $f_2^{-1} \circ I_1$ with $D_{g^{-1}} = D_{f_1} = Q_{S_1}$ and $R_{g^{-1}} = R_{f_2^{-1}} = Q_{S_2}$. Thus, $g$ is bijective, so that $S_1 \leq S_2$ and $S_2 \leq S_1$. Therefore, $S_1 = S_2$. ∎

**Remark.** The relations between the sets of subprocesses defined by each definition, and the mapping functions is graphically displayed in Figure 2.4.

That Definition 2.29 implies Definition 2.38 is rather obvious, as the simplicity of its proof shows. That is, for a superprocess $P$, the set of subprocesses of $P$ defined by Definition 2.29 is a subset of the set of subprocesses defined by Definition 2.38.

However, that Definition 2.38 in its turn implies Definition 2.29 is more delicate. What this says is that if we have a subprocess defined by Definition 2.38, then we can always find a mapping, $g(\cdot)$, to rename the states of that subprocess so that Definition 2.29 is satisfied. In this sense, for every subprocess $S_2$ defined by Definition 2.38 there exists a subprocess $S_1$ of $P$, defined by Definition 2.29, such that $g(Q_{S_2}) = Q_{S_1}$. This means that $g(S_2) \leq S_1$ and $S_1 \leq g^{-1}(S_2)$, with subprocess defined according to Definition 2.29. And thus, the set of different subprocesses defined by Definition 2.38 is not larger than the set of subprocesses given by Definition 2.29.

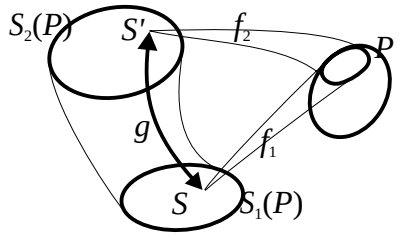



**Figure 2.4:** The relation between the different definitions of subprocesses of a process $P$, and the mapping functions. The set $\mathcal{S}_1(P)$ is the set of subprocesses defined by Definition 2.29, while $S_2(P)$ is the set defined by Definition 2.38. $f_1$, $f_2$ and $g$ are the mapping functions.

To rephrase, Lemma 2.39 means that, with a given superprocess $P$, for any subprocess $S_2$ as defined by Definition 2.38 we can always find another subprocess, $S_1$, of $P$, that satisfies both definitions and is such that $S_1 = S_2 \Leftrightarrow S_1 \leq S_2 \wedge S_2 \leq S_1$ with subprocess defined by Definition 2.38. Thus, $S_1$ and $S_2$ are *isomorphic* in the sense that they differ only in the way the states are named. But, as also pointed out by Arnold (1994), these names have no particular meaning and can be arbitrarily chosen. It is by their structural properties that transition machines are distinguished. However, for practical purposes the naming of the states can contribute considerably to the interpretation of the traces. For instance, when composing two transition machines in parallel, the state-space of the composed process becomes the cross product of the state-spaces of the respective processes, and thus a state in the composed process can immediately tell us which actions have been executed by the respective process. □

The equivalence of the two subprocess definitions has the implication that, without loss of generality, a subprocess that satisfies Definition 2.38 can always be considered to satisfy Definition 2.29, and vice versa.

### 2.3.2 Refinement

The subprocess relation is a partial ordering in which two processes are comparable if they have the same alphabet, and the structural appearance of one of them, the subprocess, is included in the other, the superprocess. This is a rather strong relation, in the sense that the number of different subprocesses of a given process is very limited. A subprocess can be nondeterministic, for instance, only if the superprocess is. In this section we will define a weaker ordering relation between transition machines, *refinement*, that allows a much larger number of different processes to be related to a given process. For instance, a nondeterministic process can refine a deterministic one.

#### Definition 2.40 Refinement

For two transition machines $S = (Q_S, \Sigma_S, I_S, M_S, E_S)$ and $P = (Q_P, \Sigma_P, I_P, M_P, E_P)$, with $\Sigma_S = \Sigma_P$, $S$ is said to *refine* $P$ if there exists a function $f\colon Q_S \to Q_P$, such that for

all traces $t_S \in E_S^*$ there exists a trace $t_P \in E_P^*$ such that

$$\begin{aligned} f(first(t_S)) &= first(t_P) \\ label(t_S) &= label(t_P) \\ f(last(t_S)) &= last(t_P) \end{aligned} \tag{2.44}$$

and

$$\forall q \in Q_S \ \begin{cases} q \in I_S \Leftrightarrow f(q) \in I_P \\ q \in M_S \Leftrightarrow f(q) \in M_P \end{cases} . \tag{2.45}$$

**Remark.** This definition is different from the subprocess definition, Definition 2.38, in that the function $f$ is not required to be injective. In fact, it is simply required to be a (total) *function*, meaning that for every state of $Q_S$, there is a unique state in $Q_P$. However, some states of $Q_S$ may be mapped onto the *same* state of $Q_P$. Thus, $f$ can be *many-to-one*, so that a nondeterministic $S$ can refine a deterministic $P$, see Figure 2.6. It is precisely this fact that will be exploited when generating a supervisor for a given process. Obviously, when $S$ is a subprocess of $P$, $S$ also refines $P$. Furthermore, when $S$ refines $P$, the language of $S$ is a subset of the language of $P$.

The definition of refinement means that all related traces of $S$ have similar related traces in $P$, and thus $L(S) \subseteq L(P)$. This also holds when $S$ is a subprocess of $P$, but then, additionally, nonrelated traces in $S$ have similar nonrelated traces in $P$. With refinement, the traces of $P$ are not necessarily unique with respect to the traces of $S$. This has the consequence that the number of traces of $P$ may be smaller than the number of traces of $S$, thus making it possible for a nondeterministic $S$ to refine a deterministic $P$. This is illustrated in Figure 2.5, see $S_2$ and $P_2$. Note also that, when $S$ refines $P$, two nonrelated traces of $S$ may or may not be related in $P$. This is also shown in Figure 2.5.

Finally we note that, when a deterministic process $S$ refines $P$, $S$ is a subprocess of $P$. This follows from the fact that when $S$ is deterministic, there does only exist one trace with a given label, so that the function $f$ must be one-to-one, see Figure 2.6. □

The following lemma will be important when showing that given a transition machine $S$ to act as a controlling unit for another transition machine $P$, we can find the "controllable part" of $S$.

**Lemma 2.41** Refinement is a *transitive* relation, that is, for transition machines $S$, $P$, and $Q$, if $S$ refines $P$ and $P$ refines $Q$ then $S$ refines $Q$.

**Proof.** When $S$ refines $P$ and $P$ refines $Q$, then there exist functions $f_1 : Q_S \to Q_P$ and $f_2 : Q_P \to Q_Q$ as defined by Definition 2.40. Obviously, the composition $f_2 \circ f_1 : Q_S \to Q_Q$ exists and satisfies Definition 2.40. Therefore, $S$ refines $Q$. ■

Refinement serves the purpose of characterizing when the composition of two transition machines will be equal to one of them, as will be shown in Section 2.4. Observe that, when considering only the accessible parts of the transition machines, the function $f$ of Definition 2.40 can be defined from the tracesets, $tr(S)$ and $tr(P)$, instead of the set of all possible traces given by $E_S^*$ and $E_P^*$. This is also the case for the subprocess relation,

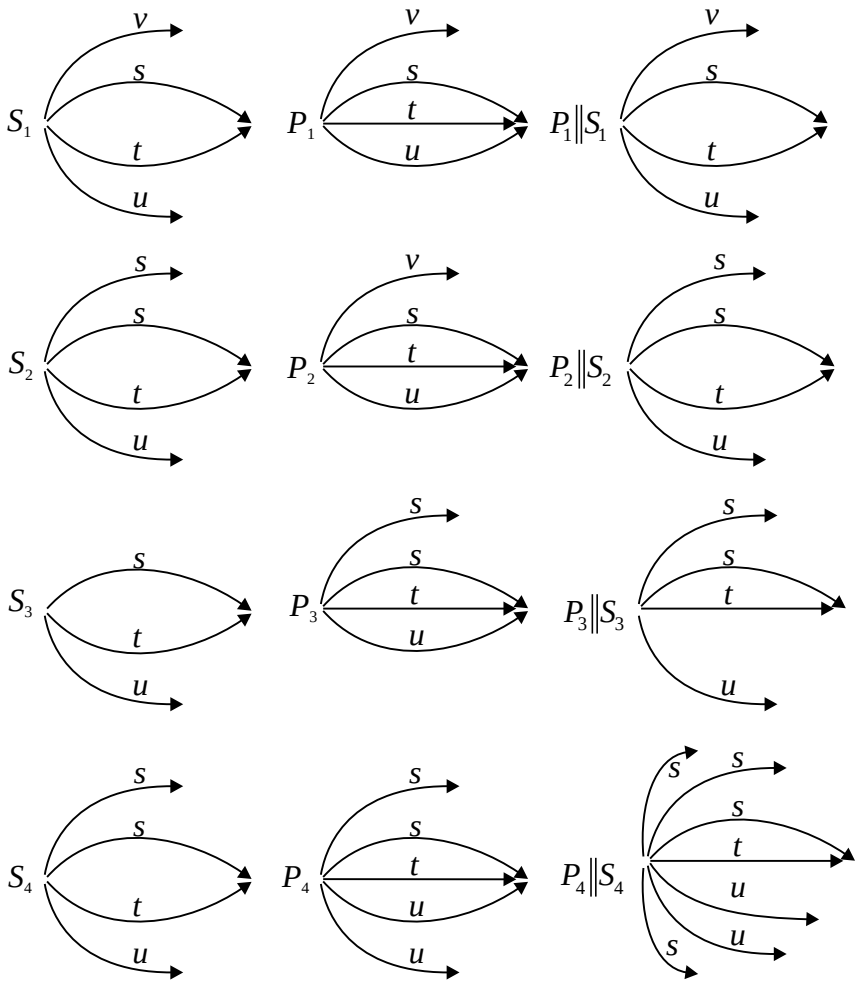



**Figure 2.5:** A number of transition machines, $P_i$, and examples of refining transition machines, $S_i$. Only the labels of the traces have been indicated. Note that $P_2$ is deterministic, while $S_2$ is nondeterministic, yet still $S_2$ refines $P_2$. To the right is shown the full synchronous composition of the respective pair of automata. We can also note that each $S_i$ is a subprocess of the respective $P_i \parallel S_i$, as well as $S_4$ is a subprocess of $P_4$.

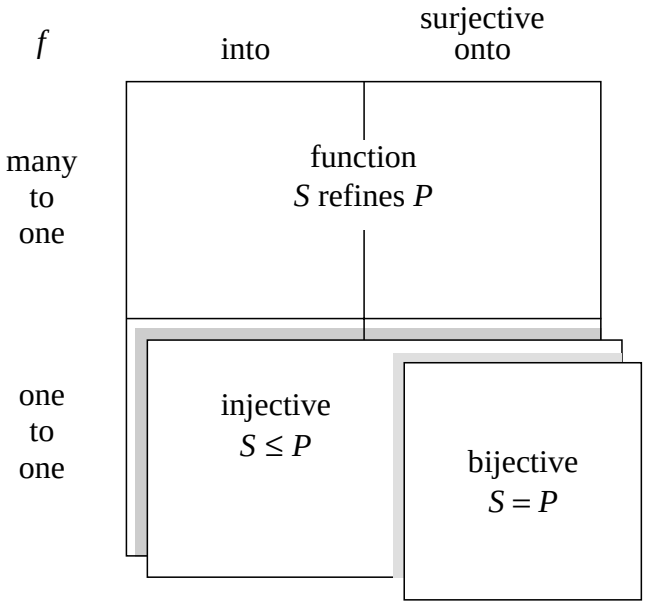



**Figure 2.6:** The relation between refinement and subprocess. $S$ refines $P$ when there exists a function $f$ mapping the states of $Q_S$ into $Q_P$. When this function is injective, $S$ is a subprocess of $P$, and when this function is bijective, lower right quadrant, $S$ and $P$ are equal.

and it is a consequence of the fact that when a transition machine is accessible, its state-space is defined by its traceset, see Lemma 2.16. Quite commonly, we will only regard the accessible part of a given transition machine, and the largest accessible part, at that. This is well-defined, as will be shown in Chapter 4.

### Summary

In this section we have defined ordering relations between transition machines, refinement and subprocesses. Refinement is coarser than the subprocess relation in the sense that, for a given transition machine, the number of its refining processes is much larger than the number of its subprocesses within the universe of discourse. For instance, a deterministic process can have nondeterministic refining processes but only deterministic subprocesses.

It has also been shown that the condition on the state-space of the intuitive notion of a subprocess can be relaxed, and subprocesses can be defined by an injective mapping from the state-space of the subprocess to the state-space of its superprocess. When two transition machines are equivalent there exists a bijective mapping. This is important since we will in the next section show that, when a process $S$ refines a deterministic process $P$, their concurrent execution with events in both their alphabets executed by both processes or none (synchronization), is equal to $S$. Naturally, the state-space of the process describing the interaction of $P$ and $S$ is not a subset of any of the processes, and particularly not of $Q_S$.

## 2.4 Operations On Processes

Operations on processes can be divided in two sections; one defining operations on processes in general, and one defining operations on subprocesses. However, since the universe of discourse is equal for all processes, any pair of transition machines can be considered to be subprocesses of some process. Nonetheless, we will make the distinction here, since the operations on subprocesses will only be used in specific contexts.

### 2.4.1 General Operations

The general operations concern, in fact, only one operator with two important specializations. These model processes executing under various forms of interaction. The composition results in a new transition machine that expresses the behavior that is the result of the interaction. Obviously, the various forms of interaction model different constraints of synchronization, from *interleaving* that includes no constraints, to *full synchronization* that requires all mutual events to synchronize. The general operator is called *parallel composition with partial synchronization*, abbreviated PCPS.

**Definition 2.42 Parallel Composition**

For two transition machines $P = (Q_P, \Sigma_P, I_P, M_P, E_P)$ and $S = (Q_S, \Sigma_S, I_S, M_S, E_S)$ we define *parallel composition with partial synchronization*, denoted $P \|_A S$, with $A \subseteq \Sigma_P \bigcap \Sigma_S$, as

$$P\|_A S = (Q_P \times Q_S, \Sigma_P \bigcup \Sigma_S, I_P \times I_S, M_P \times M_S, E_{P\|_A S}) \tag{2.46}$$

where

$$\forall (p,\sigma,p') \in E_P \ \forall (q,\tau,q') \in E_S \begin{cases} ((p,q),\sigma,(p',q)) \in E_{P\|_A S} & \sigma \notin A \\ ((p,q),\tau,(p,q')) \in E_{P\|_A S} & \tau \notin A \\ ((p,q),\sigma,(p',q')) \in E_{P\|_A S} & \sigma = \tau \in A \end{cases} . \tag{2.47}$$

**Remark.** Note that, when composing two (accessible) transition machines in parallel under partial synchronization not all states of the composed process necessarily become accessible. The set $A$ is referred to as the *synchronization set* (or synchronization alphabet). □

PCPS is also defined by Heymann (1991) where a number of its properties are shown. For our present purposes it suffices to recognize the following two specializations of PCPS, *synchronous composition* and *interleaving*.

**Definition 2.43 Synchronous Composition**

The (full) *synchronous composition* of two transition machines $P$ and $S$, denoted by $P \| S$, is their parallel composition with the intersection of their alphabets as the synchronization set. That is,

$$P\| S = P\|_{\Sigma_P \bigcap \Sigma_S} S \tag{2.48}$$

**Remark.** Thus, the full synchronous composition of two transition machines *requires* all mutual events to synchronize. An event that is in both machines' alphabets can only occur when both processes agree upon executing it. Non-mutual events, on the other hand, can be executed whenever the respective transition machine is in a state to do so. The transition function for the composed system is thus

$$\delta_{P\|S}(\langle p,q\rangle,\sigma) = \begin{cases} \delta_P(p,\sigma)\times\delta_S(q,\sigma) & \sigma\in\Sigma_P\bigcap\Sigma_S \\ \delta_P(p,\sigma)\times\{q\} & \sigma\in\Sigma_P-\Sigma_S \\ \{p\}\times\delta_S(q,\sigma) & \sigma\in\Sigma_S-\Sigma_P \end{cases} \tag{2.49}$$

When $\Sigma_P=\Sigma_S$, every event occurs if and only if both processes are ready to execute it. The edge-set of the composed transition machine becomes

$$E_{P\|S} = \Big\{(\langle p,q\rangle,\sigma,\langle p',q'\rangle)\,|\langle p',q'\rangle\in\delta_{P\|S}(\langle p,q\rangle,\sigma)\Big\}. \tag{2.50}$$

□

When composing two transition machines under full synchronous composition, the language of the resulting automaton can be readily defined. However, for our purposes only the following special cases will be considered.

**Lemma 2.44** For two transition machines $S$ and $P$ with $\Sigma_S\subseteq\Sigma_P$, let $\Sigma_{P-S}=\Sigma_P-\Sigma_S$ be the set of events of $\Sigma_P$ not in $\Sigma_S$ and let $\Sigma^*_{P-S}$ denote all finite strings over $\Sigma_{P-S}$ including the empty string. Then the following language inclusions hold,

$$L(P)\bigcap L(S)\Sigma_{P-S}\subseteq L(P)\bigcap L(S)\Sigma^*_{P-S}\subseteq L(P\,\|\,S). \tag{2.51}$$

Here $L(S)\Sigma_{P-S}$ and $L(S)\Sigma^*_{P-S}$ denotes the languages obtained by concatenating all strings of $L(S)$ with all elements of $\Sigma_{P-S}$ and $\Sigma^*_{P-S}$, respectively.

**Proof.** We will show this by induction on the length of a string $s=\sigma_0\sigma_1\ldots\sigma_n\in L(P)$. We will show only the rightmost inclusion. The left one follows immediately.

$(n=0)$ When $n=0$, $s$ is the empty string and the inclusion trivially holds. All languages are prefix closed.

$(n=m)$ We assume that the inclusion holds for this case, that is, $s_m=\sigma_0\ldots\sigma_m\in L(S)\Sigma^*_{P-S}\bigcap L(P)$ and $s_m\in L(P\|\,S)$.

$(n=m+1)$ For $s_m\sigma_{m+1}$ there are three possibilities with regard to $L(S)$.

1. $\sigma_{m+1}\in\Sigma_{P-S}$ so that $\sigma_{m+1}\notin\Sigma_S$ and thus $s_m\sigma_{m+1}\notin L(S)$. Then $P$ can execute $\sigma_{m+1}$ after $s_m$ as it pleases. The event $\sigma_{m+1}$ cannot be prevented by $S$ from occuring in $P$. Thus, $s_m\sigma_{m+1}\in L(P\|\,S)$.

2. $\sigma_{m+1}\in\Sigma_S$ and $s_m\sigma_{m+1}\in L(S)$. Then both $S$ and $P$ can agree on executing $\sigma_{m+1}$ after $s_m$, and $s_m\sigma_{m+1}\in L(P\|\,S)$.

3. $\sigma_{m+1}\in\Sigma_S$ but $s_m\sigma_{m+1}\notin L(S)$. In this case $S$ prevents $P$ from executing $\sigma_{m+1}$ after $s_m$, and $s_m\sigma_{m+1}\notin L(P\|\,S)$.

So, whenever a string $ss' \in L(P)$ and either $s \in L(S)$ and $s' \in \Sigma_{P-S}^*$ or $ss' \in L(S)$, then $ss' \in L(P \| S)$, since when $s' \in \Sigma_{P-S}^*$ then $ss' \in L(S)\Sigma_{P-S}^*$. Thus Lemma 2.44 holds. ■

**Remark.** We can view this as a case where $S$ has no *control* over the events of $\Sigma_{P-S}$. These events are *uncontrollable* to $S$, so that $P$ can execute them whenever in a state to do so. This concept of uncontrollable events is in fact a crucial issue for the supervisory control theory, which will be presented in Chapter 3. □

The following corollary to Lemma 2.44 follows from the fact that when $\Sigma_S = \Sigma_P$, we have that $\Sigma_{P-S} = \emptyset$, and it tells us what language results when two automata with equal alphabets are fully synchronized.

**Corollary 2.45** For two transition machines $P$ and $S$ with $\Sigma_P = \Sigma_S$, the following holds

$$L(P \| S) = L(P) \bigcap L(S) \tag{2.52}$$

In a similar manner it can be shown that when $\Sigma_S = \Sigma_P$, then ready set of any state of the composed system, $P \| S$, is the intersection of the ready sets of the corresponding states of the original processes. That is

$$\forall \langle p, q \rangle \in Q_{P\|S} \; out(\langle p, q \rangle) = out(p) \bigcap out(q). \tag{2.53}$$

The requirement that $\Sigma_S = \Sigma_P$ is in fact a loss of generality, and it is really unnecessary. Equation (2.53) holds (with a slight reformulation) even if this constraint is relaxed. However, the proof becomes more intricate involving *restriction* (see Hoare (1985)) of strings to the alphabets of the respective processes, a notion which we have chosen not to define, simply because we will not need it. The assumption that $\Sigma_S = \Sigma_P$ will be globally introduced, and it is a common assumption within the supervisory control theory.

A special case is worth mentioning though, since it will be used to derive requirements for a supervisor process to be able to control another process. This is the case when $\Sigma_S \subseteq \Sigma_P$. Then the following holds,

$$\forall \langle p, q \rangle \in Q_{P\|S} \; out(\langle p, q \rangle) = out(p) \bigcap [out(q) \bigcup \Sigma_{P-S}] \tag{2.54}$$

where $\Sigma_{P-S} = \Sigma_P - \Sigma_S$, as in Lemma 2.44. Since we always have that the active set of a string is the union over the ready sets of the states reached by that string, Lemma 2.44 follows from (2.54), as is readily verified. Usually, when two processes are brought together to evolve concurrently, the intention is that they are to interact. Such interactions are then expressed by mutual events on which the processes synchronize. This is what the synchronous composition models. However, sometimes it is more useful to join processes with mutual events to operate concurrently, without directly interacting with each other. Such a composition of transition machines is called *interleaving*.

### Definition 2.46 Interleave

The *interleaving* of two transition machines $P$ and $S$, is their parallel composition with the empty set as the synchronization set. That is,

$$P \|_{\emptyset} S \tag{2.55}$$

**Remark.** Interleaving is similarily defined by Hoare (1985), though for *failure model* systems (see Section 2.5). The interleaving of two processes *prohibits any* mutual events to synchronize. Thus, any process can execute any event irrespective of the other process, even though this event may exist in the alphabet of both processes. Each action of the interleaved system is an action of exactly *one* of the processes. When an event occurs, if one of the processes could not have executed the event it must have been the other. However, if both processes could have engaged in the event, then the choice between them is nondeterministic. Therefore, interleaving two deterministic processes with (partly) mutual alphabets, results in a nondeterministic automaton; that is, an automaton in whcih one and the same string can lead to any of a number of states.

When interleaving two accessible transition machines, all states of the interleaved process become accessible.

□

In Figure 2.7 the different ways two simple transition machines can be composed in parallel under partial synchronization, is shown. Note that the alphabets of the two automata are only partially mutual, meaning that some events, $a$ and $b$, can *never* synchronize. Note also that $P$ includes the $e$ event in its alphabet, but there is no transition labeled by $e$. Thus, whenever synchronization on $e$ is requested, as in $\Sigma_1$, for instance, $e$ is hindered from ever occuring.

**Lemma 2.47** For any three transition machines $P$, $Q$, $R$, PCPS is associative. That is

$$P \parallel_A (Q \parallel_B R) = (P \parallel_A Q) \parallel_B R, \tag{2.56}$$

whenever $A = B$ or $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$,

**Proof.** The proof is cumbersome and given in Appendix B. ■

**Remark.** This may seem a very special case. However, the algorithm to be presented relies on the associativity of PCPS under these conditions. Note that $A = B$ includes interleaving, $A = B = \emptyset$, and when $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$ the operation is full synchronous composition. Observe that PCPS is not associative when, for instance $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \emptyset$. An annoying fact which will be commented upon later. □

In the following theorem we will consider only the accessible parts of the given and composed transition machines. We have hitherto tried to keep the propositions on as general a level as possible, not requiring the transition machines to be accessible. However, for practical purposes the restriction to the accessible parts has no consequences; the behavior of a transition machine can only ever concern its accessible states. We will introduce the requirement of the transition machine $P$ to be deterministic though. This restricts the generality of the given lemma, but in the next chapters $P$ will represent the plant to be controlled. We will assume that such a plant is always modeled as a deterministic automaton, an assumption that we feel is adequate for most purposes. See Shayman (1994) and Overkamp (1995), though, for discussions concerning non-deterministic models representing uncertainty of the physical plant behavior.

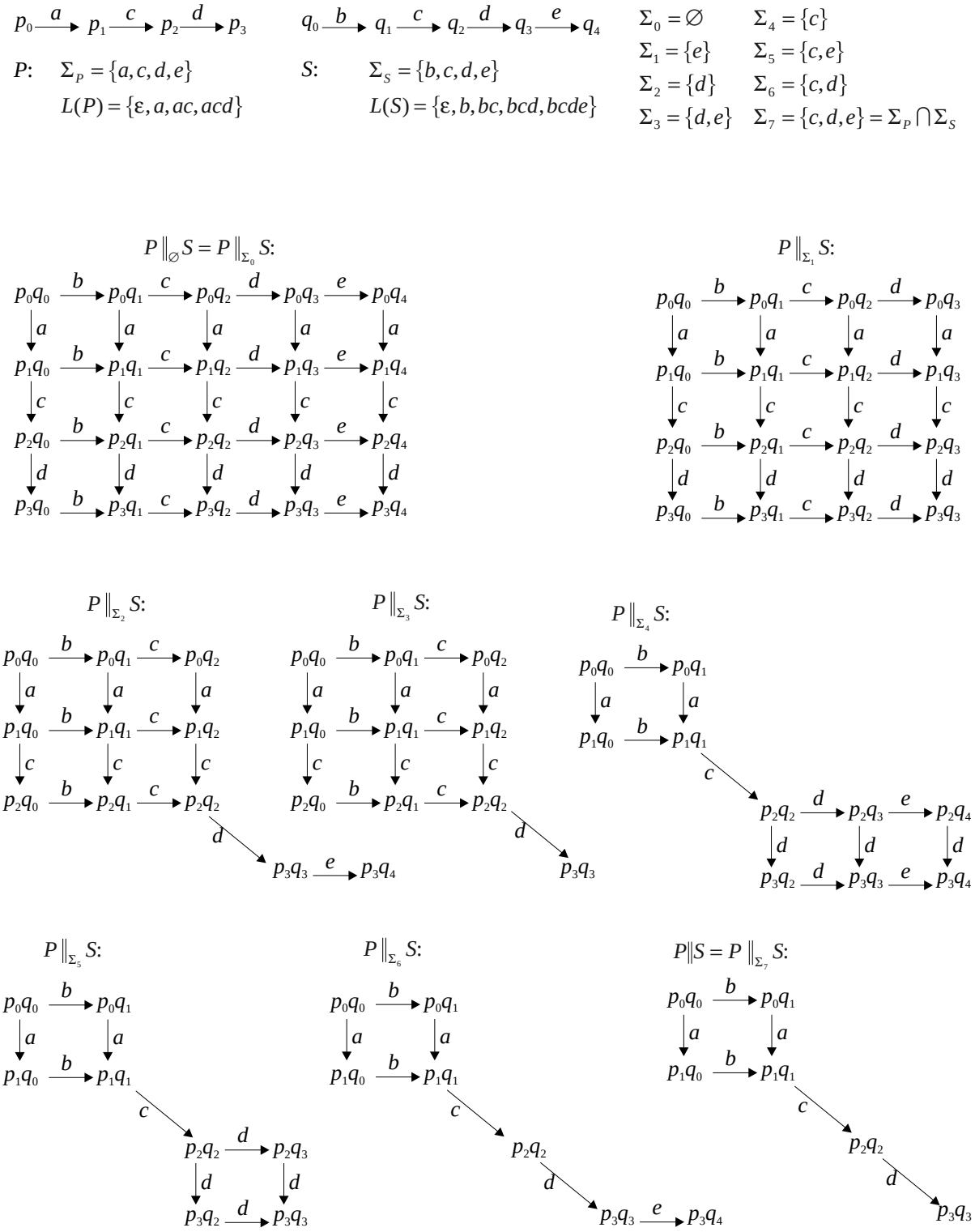



**Figure 2.7:** The eight possible accessible transition machines resulting from composing the $P$ and $S$ automatons shown at the top. The different composition alphabets are shown top right.

**Theorem 2.48** For two transition machines $S$ and $P$, with $P$ deterministic, when $S$ refines $P$ their synchronous composition equals $S$. That is,

$$P \text{ deterministic} \Rightarrow (S \text{ refines } P \Leftrightarrow P \| S = S), \tag{2.57}$$

when we only consider the largest accessible parts of the transition machines.

**Proof.** When we only consider the accessible parts of the transition machines, the functions defining refinement and equality of processes, can be defined from the tracesets of the respective machines. That is, $S$ refines $P$ if there exists a function $f: Q_S \to Q_P$ mapping such that for all traces $t_S \in tr(S)$ there exists a trace $t_P \in tr(P)$, such that (2.44) holds. Similarily, equality between $P \| S$ and $S$ can be defined by a bijective function from the state-space $Q_{P\|S}$ to $Q_S$, such that for all traces of $tr(P \| S)$ there exists traces of $tr(S)$ satisfying (2.42).

By definition, the full synchronous composition of two transition machines result in similar traces being composed such that the composed trace' $last(\cdot)$ equals the state in the composed automaton given by the ordered pair of the original traces' $last(\cdot)$. That is, for $t_S \in tr(S)$ and $t_P \in tr(P)$, when $t_S \; sim \; t_P$, there exists a trace $t_{P\|S} \in tr(P \| S)$ such that $last(t_{P\|S}) = \langle last(t_P), last(t_S) \rangle$. For a nondeterministic transition machine $S$, two traces $t_S, t'_S \in tr(S)$ can be different, yet have equal labels, that is, $t_S \neq t'_S$ but $t_S \; sim \; t'_S$. This cannot be the case for a deterministic transition machine. With a deterministic machine, two traces are similar if and only if they are equal.

($\Rightarrow$) It is easy to see that there does always exist a function $f: Q_{P\|S} \to Q_S$ such that $P \| S$ refines $S$. Choose, for instance, the function that picks out the elements of $Q_S$ from the ordered pairs of $Q_{P\|S}$. That is, for $\langle p, q \rangle \in Q_{P\|S}$, we have that $f(\langle p, q \rangle) = q$. This follows from the definition of the synchronous composition.

When $S$ refines $P$, every trace of $S$ can be mapped onto a trace of $P$ by a function $f_S: Q_S \to Q_P$. Say that we have the traces $t_S, t'_S \in tr(S)$ and $t_P, t'_P \in tr(P)$, such that (with some abuse of notation) $f_S(t_S) = t_P$ and $f_S(t'_S) = t'_P$. To simplify our notation, define $p = last(t_P)$, $p' = last(t'_P)$, $q = last(t_S)$, and $q' = last(t'_S)$. Since for these traces $t_S \; sim \; t_P$ and $t'_S \; sim \; t'_P$, we have that there exists traces $t_{P\|S}, t'_{P\|S} \in tr(P \| S)$ such that $t_{P\|S} \; sim \; t_S$ and $t'_{P\|S} \; sim \; t'_S$. Furthermore, $last(t_{P\|S}) = \langle p, q \rangle$ and $last(t'_{P\|S}) = \langle p', q' \rangle$. When two traces of $S$ are related, their corresponding traces in $P$ must also be related, otherwise $f_S$ is not a function. This means that $\langle p, q \rangle = \langle p', q' \rangle$ if and only if $q = q'$. Therefore, the function $f$ is injective, meaning that $f(\langle p, q \rangle) = f(\langle p', q' \rangle) \Rightarrow \langle p, q \rangle = \langle p', q' \rangle$. Thus $P \| S \leq S$.

Furthermore, when $P$ is deterministic there exists only one trace with a given label so that when $t_S \; sim \; t'_S$ then $t_P = t'_P$., with $t_S$, $t'_S$, $t_P$, $t'_P$ as defined in the above paragraph. Therefore, two similar but non-equal traces of $P \| S$ can only differ in the "$S$-part". This means that for every trace of $S$ there will exist a trace in $P \| S$, and all traces of $P \| S$ will have corresponding traces in $S$. Therefore, the function $f$ is injective (as above) and onto, that is, the domain of $f$ is equal to the entire state-space $Q_S$. This makes $f$ bijective, so that $P \| S = S$.

($\Leftarrow$) When $f: Q_{P\|S} \to Q_S$ is bijective, its inverse $f^{-1}: Q_S \to Q_{P\|S}$ exists. Let us define the function $f_P: Q_{P\|S} \to Q_P$ such that $f_P(\langle p, q \rangle) = p$. Then the composite function

$(f_P \circ f^{-1})$:$Q_S \to Q_P$ exists. However, when $P$ is deterministic, and $S$ nondeterministic, for two states reached by the same string in $P \,\|\, S$, and hence in $S$, only one state is reached in $P$. Thus, for $t_{P\|S}$ *sim* $t'_{P\|S}$, we have that $f_P(last(t_{P\|S})) = f_P(\langle p_1, q_1\rangle) = f_P(last(t'_{P\|S})) = f_P(\langle p_1, q_2\rangle) = p_1$, so that $f_P$ is not injective and therefore $f_P \circ f^{-1}$ is not injective. Thus, $S$ refines $P$. ■

**Remark.** From the proof of Theorem 2.48 it follows that the synchronous composition $P \,\|\, S$ always refines $P$ (as well as $S$). This holds irrespective of whether $S$ refines $P$ or not, and this is important since when $P$ is deterministic then $P\|\,(P\|\,S) = P\|\,S$, by Theorem 2.48. This means that a nondeterministic specification $S$ can be composed under full synchronization with a deterministic plant $P$, and their composition can be used as a supervisor for the plant. (Assuming some other properties are upheld, such as completeness, see Section 3.4.) A similar result for deterministic processes was given by Kumar (1991). □

In Figure 2.5, on page 43, examples of the synchronous composition between a refining process and the refined process, are given. As can be seen, both $P_1\|\,S_1$ and $P_2\|\,S_2$ equal the refining processes $S_1$ and $S_2$, respectively. This so, since $P_1$ and $P_2$ are both deterministic. It can also be seen that the resulting synchronization of a refining process, $S_i$, with the refined process, $P_i$, really refines $P_i$.

When a transition machine is a subprocess of another machine, then it also refines its superprocess. Thus, Theorem 2.48 has the following important special case.

**Corollary 2.49** For two transition machines $S$ and $P$, with $P$ deterministic, if $S$ is a subprocess of $P$ then their synchronous composition equals $S$. That is,

$$P \text{ deterministic} \Rightarrow (\ S \leq P \Rightarrow P\|\,S = S), \tag{2.58}$$

when we only consider the largest accessible parts of the transition machines.

**Remark.** We can note that, when $S \leq P$ and $P$ is determinstic, so must also $S$ be. It is always the case that when $P$ is deterministic $P \,\|\, S \leq S$, and when $S$ is deterministic, $P \,\|\, S \leq P$. However, when $P$ and $S$ are both allowed to be nondeterministic, their synchronous composition only refines $P$ and $S$, respectively. It is easy to see that the functions $f_P(\langle p, q\rangle) = p$ and $f_S(\langle p, q\rangle) = q$ determine this. □

### 2.4.2 Operations On Subprocesses

The parallel composition with partial synchronization makes sense for modeling the interaction of arbitrary transition machines. However, when studying the partial ordering of subprocesses of a given superprocess, the operations of *subprocess union* and *subprocess intersection* are more relevant.

**Definition 2.50 Subprocess Union and Intersection**

For two subprocesses $Q$ and $R$ of a superprocess $P$, their *union* is defined as

$$Q \bigcup R = (Q_Q \bigcup Q_R, \Sigma_P, I_Q \bigcup I_R, M_Q \bigcup M_R, E_Q \bigcup E_R), \tag{2.59}$$

and their intersection is defined as

$$Q \bigcap R = (Q_Q \bigcap Q_R, \Sigma_P, I_Q \bigcap I_R, M_Q \bigcap M_R, E_Q \bigcap E_R). \tag{2.60}$$

**Remark.** Note that subprocess union and subprocess intersection join and intersect the respective sets of the 5-tuples representing the processes. The alphabet will in either case be the same as the alphabet of the superprocess, here $\Sigma_P$. □

When intersecting two subprocesses, it may happen that their state-spaces and edge-sets are disjoint, so that the state-space and the edge-set of the resulting process are empty. For this to be a valid result of subprocess intersection, we make the following definition, which will also come in handy when attempting to prove existence of certain types of subprocesses.

**Definition 2.51 Null Process**

The smallest subprocess of any transition machine $P$ is the *null process*, defined as

$$\emptyset_P = (\emptyset, \Sigma_P, \emptyset, \emptyset, \emptyset). \tag{2.61}$$

**Remark.** Note that the null process is defined for any transition machine, and that it has that transition machines alphabet. Hence the index $_P$. When denoting the null process corresponding to *any* transition machine, this index is omitted. Naturally, $tr(\emptyset) = \emptyset$ and $L(\emptyset) = \emptyset$.

By the definitions of accessibility, Definition 2.15, and coaccessibility, Definition 2.17, and since the null process has no states it trivially holds that the null process is always accessible and coaccessible. Therefore, it is also trim. □

There are a number of properties of subprocess union and subprocess intersection that are important for the synthesis of a supervisor described in Chapter 4. These are summarized in the following theorem. Since a process is described as an ordered 5-tuple of sets, and since subprocess union and subprocess intersection unite and intersect the respective sets of the 5-tuple, Theorem 2.52 follows immediately from the definitions. Therefore, the following theorem is given without proof.

**Theorem 2.52** For two subprocesses $Q$ and $R$ of a superprocess $P$, the following holds,

1. $Q \leq P \wedge R \leq P \Rightarrow Q \bigcup R \leq P \wedge Q \bigcap R \leq P$
2. $Q \bigcup R = R \Leftrightarrow Q \leq R \Leftrightarrow Q \bigcap R = Q$
3. $R \leq Q \bigcap P \Leftrightarrow R \leq Q \wedge R \leq P$
4. $R \bigcup Q \leq P \Leftrightarrow R \leq P \wedge Q \leq P$

**Remark.**

1. Since the union and intersection of two arbitrary subprocesses of a given superprocess, are still subprocesses of that superprocess, the set of subprocesses of a given superprocess is closed under both subprocess union and subprocess intersection. The union of any number of subprocesses can never be "larger" than the superprocess. This is a significant fact that will be used when proving that there does exist unique maximal elements of sets of subprocesses of a given process, in Section 4.1.

2. For two arbitrary processes, one is a subprocess of the other, if and only if their intersection is equal to one of them, and if and only if their union is equal to one of them. Then their intersection is the subprocess, and their union is the superprocess.

3. When a process is a subprocess of a superprocess that can be written as the intersection of two processes, then the subprocess is also a subprocess of both these processes.

4. When the union of two processes is a subprocess of some superprocess, both of these processes are subprocesses of that superprocess.

The last two of these propositions will be of great importance when we examine efficient ways to calculate specific subprocesses of a given process. □

When composing subprocesses under union and intersection we know that the composed processes are still subprocesses of the same superprocess. Furthermore, the tracesets of the composed processes hold a number of properties relative to the trace ets of their original processes. Some of these properties are shown in the following lemmas.

**Lemma 2.53** For two subprocesses $Q$ and $R$ of a superprocess $P$, we have that

$$tr(Q) \bigcup tr(R) \subseteq tr(Q \bigcup R). \tag{2.62}$$

**Proof.** Assume that Lemma 2.53 does not hold. That is, there exists a trace $t \in tr(Q) \bigcup tr(R)$ such that $t \notin tr(Q \bigcup R)$. By definition $t \in tr(Q) \bigcup tr(R) \Leftrightarrow t \in E_Q^* \bigcup E_R^* \wedge first(t) \in I_Q \bigcup I_R$ and $t \notin tr(Q \bigcup R) \Leftrightarrow t \notin (E_Q \bigcup E_R)^* \vee first(t) \notin I_Q \bigcup I_R$. That $first(t) \in I_Q \bigcup I_R$ and at the same time $first(t) \notin I_Q \bigcup I_R$ is an obvious contradiction. However, since $E_Q^* \bigcup E_R^* \subseteq (E_Q \bigcup E_R)^*$ we also have a contradiction in that $t \in E_Q^* \bigcup E_R^* \subseteq (E_Q \bigcup E_R)^*$ and that $t \notin (E_Q \bigcup E_R)^*$. Thus all traces of $Q$ and all traces of $R$ are included in the traces of $Q \bigcup R$. ■

**Lemma 2.54** For two subprocesses, $Q$ and $R$ of a superprocess $P$, we have that

$$E_Q^* \bigcap E_R^* = (E_Q \bigcap E_R)^*. \tag{2.63}$$

**Proof.** The set $E_Q^*$ contains all (finite) traces composed of the edges of $E_Q$, and the same holds for $E_R^*$ and $E_R$. $E_Q^* \bigcap E_R^*$ contains all traces composed of edges both in $E_Q$ and $E_R$, that is, edges in $E_Q \bigcap E_R$. This must of course be the same set as $(E_Q \bigcap E_R)^*$. ■

From Lemma 2.54 and the fact that the traces of the traceset always start at initial states, it immediately follows that the traceset of a subprocess composed by intersecting two subprocesses of a given superprocess, is equal to the intersection of the tracesets of the original processes.

**Lemma 2.55** For two subprocesses $Q$ and $R$ of a superprocess $P$, we have that

$$tr(Q) \bigcap tr(R) = tr(Q \bigcap R). \tag{2.64}$$

**Remark.** Similarily it can be shown that $tr_m(Q \bigcap R) = tr_m(Q) \bigcap tr_m(R)$. Also, from Lemma 2.53 it can similarily be shown that $tr_m(Q) \bigcup tr_m(R) \subseteq tr_m(Q \bigcup R)$. □

**Summary**

In this section we have defined an operator for composing transition machines in parallel. The result is another transition machine that models the concurrent execution of the original processes under the synchronization constraint given by the synchronization alphabet. Two important specializations of this operator, full synchronous composition and interleaving, have been defined, and conditions have been given under which these operations are associative. In the Chapter 3 these two operations will be used to compose a transition machine from a number of given processes. This composed transition machine will describe a specification on the behavior of a given system, a *specification* from which a supervisor, controlling the system will be calculated.

The fact that the union of two subprocesses of a given superprocess is still a subprocess of that superprocess will be used extensively in showing that a maximal subprocess satisfying some characterizations does exist. These characterizations are for the subprocess to be trim and complete with respect to the plant to be controlled. These are all characterizations that we will require a supervisor to satisfy. In efficiently calculating this maximal subprocess the properties shown by Theorem 2.52 will be used.

## 2.5 Transition Machine Modeling Approaches

Several approaches to modeling discrete event processes have been proposed in the literature, and it is interesting to briefly study some of these; those that are most related to this work. No attempt is being made, however, to be complete in citing related work. There is just too much literature available on discrete event processes.

The approaches that we will study are two that have been used in a non-deterministic supervisory control theory setting, namely the *failures model* of Hoare (1985) and the *trajectory model* of Heymann (1991). We will also make some remarks on interesting aspects of the work of Milner (1989), when discussing aspects of generated and accepted events. And finally, we give an example to justify our need for a modeling formalism that includes states. We can note that both Hoare (1985) and Milner (1989) are concerned with modeling communication between concurrently executing processes. Heymann (1991), on the other hand, aims to develop a general algebra for discrete event systems.

The two approaches can be ordered according to increasing detail of modeling. It will be shown by an example that the trajectory model distinguishes processes that are not distinguished by the failures model. This increasing detail is very significant regarding the equivalence of discrete event processes. We will try to follow this increasing detail in the following sections.

### 2.5.1 The Failures Model

The *failures model* of Hoare (1985) models discrete event processes by the events they can execute and the events they can refuse to execute. A (nondeterministic) process is described by a set of ordered pairs of a string and a *refusal set*. The refusal set is itself a set of event-sets that the process *may* refuse if some other process, the *environment*, offered them by means of the synchronous composition. The set of all ordered sets of strings and refusal sets is called the *failures* of the process. The failures of a process is more informative about the behavior of the process than its language, which can be defined in terms of the failures. The failures distinguishes between states reached by similar strings but with different ready sets, but equates states reached by similar strings and with equal ready sets. See Example 2.1 and Example 2.3.

A deterministic process is one that (in the words of Hoare (1985)) "can never refuse any event in which it can engage". This means that a nondeterministic process can, after a string $s$, both refuse and engage in some event, say $\sigma$. That is, the event $\sigma$ is both part of the active set after $s$, and part of the refusal set after $s$. The active set is not defined by Hoare (1985), claiming that the refusal sets are slightly simpler since they obey some mathematical laws, whereas the corresponding laws for the active set would be more complicated. Using the active set, though, we can write a formal expression that explains when a process is nondeterministic in the failures model. A process $P$ is nondeterministic in the failures model sense, if

$$\exists s \in L(P)\ \exists X \in \mathrm{ref}(P,s)\ \gamma(L(P),s) \bigcap X \neq \emptyset. \tag{2.65}$$

Here $\mathrm{ref}(P,s)$ represents the refusals of $P$ after string $s$.

A number of operators for composing processes are introduced by Hoare (1985). Of main interest to us are the full synchronous composition and interleave. Though the synchronous composition is introduced for processes with non-equal alphabets, to interleave two processes it is required that their alphabets be equal. Why this restriction is introduced, we have not been able to penetrate.

The main component of the theory presented by Hoare (1985) used in this thesis is *sharing by interleaving*. Hoare (1985) introduces the concept of a *named subordinate process*, whose sole task is to meet the needs of a main process. This is done by parametrizing the events of the main process, so as to distinguish the subordinate process. The parameter serves as a *local name* for the subordinate process. Suppose then that the main process consists of two processes operating in full synchrony, both of who need to communicate independently with a shared process. Then the parametrized events are not sufficient. Since these events would be parametrized equally in both of the synchronous main processes, their sharing of the subordinate process would have to be synchronous. This is not the intended effect.

**Example 2.1 Failures and Synchronization**

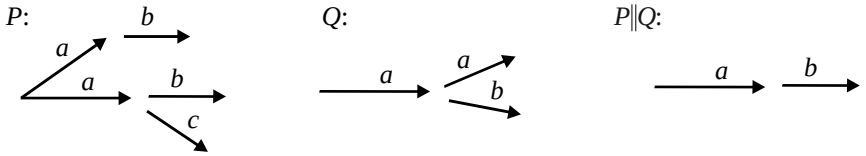



**Figure 2.8:** Two processes $P$ and $Q$, and their full synchronous composition

This example serves to illustrate the failures model. In Figure 2.8 three processes are given, $P$, $Q$ and their synchronous composition $P \| Q$. The processes have equal alphabets, $\Sigma = \{a, b, c\}$, and the failures describing these processes are

$$\begin{aligned} \text{failures}(P) &= \left\{ \langle \varepsilon, \{\emptyset, \{b\}, \{c\}, \{b, c\}\} \rangle, \langle a, \{\emptyset, \{a\}, \{c\}, \{a, c\}\} \rangle, \left\langle ab, 2^{\Sigma} \right\rangle, \left\langle ac, 2^{\Sigma} \right\rangle \right\} \\ \text{failures}(Q) &= \left\{ \langle \varepsilon, \{\emptyset, \{b\}, \{c\}, \{b, c\}\} \rangle, \langle a, \{\emptyset, \{a\}, \{c\}, \{a, c\}\} \rangle, \left\langle ab, 2^{\Sigma} \right\rangle, \left\langle aa, 2^{\Sigma} \right\rangle \right\} \\ \text{failures}(P \| Q) &= \left\{ \langle \varepsilon, \{\emptyset, \{b\}, \{c\}, \{b, c\}\} \rangle, \langle a, \{\emptyset, \{a\}, \{c\}, \{a, c\}\} \rangle, \left\langle ab, 2^{\Sigma} \right\rangle \right\} \end{aligned} \tag{2.66}$$

where $2^{\Sigma}$ as usual denotes the set of all subsets of $\Sigma$.

We can note that $P$ is nondeterministic with respect to the failures model, since $\gamma(L(P), a) \bigcap \{c\} = c$ and $\{c\} \in \text{ref}(P, a)$. Thus, $P$ can both refuse the event $c$, while it can also engage in that event. This means that there must exist two states (at least) reachable by $a$, one where $c$ is refused, and one where it is not, just as shown in Figure 2.8. We can also note the, perhaps surprising fact that $P \| Q$ is deterministic, despite $P$ being nondeterministic. This is a consequence of the fact that the two branches that would arise if we were to synchronize the state-machine representations of $P$ and $Q$, represent equivalent strings as well as equivalent refusal sets. Thus, these failures are not distinct. See also Example 2.3

■

The problem is that two processes need to independently share a mutual resource. A general method of sharing is provided by *multiple labeling*; that is, we create enough local names for the subordinate process to be shared between all sharing processes. In essence, we create extra transitions labeled by the parametrized events. However, this requires that all sharing processes are known in advance, and is not adequate for a subordinate process intended to serve the needs of an unknown number of unknown sharing processes.

This leads Hoare (1985) to *interleave* the sharing processes, before synchronizing with the subordinate process. Thus, the sharing processes can both include the event-set of the subordinate process, without any synchronization taking place between the sharing processes. Of course, this prohibits direct synchronization between the sharing processes, but only as long as full interleaving is used. With the parallel composition operator of Definition 2.42, (not defined by Hoare (1985)), this can also be achieved. Sharing by interleaving will be discussed in detail in Chapter 5.

The failures model is used by Overkamp (1995) to model non-deterministic systems, and to extend the supervisory control theory to non-deterministic plant and specification. See Section 3.4.

### 2.5.2 The Trajectory Model

The trajectory model has much in common with the failures model. Just like the failures model, the trajectory model disposes of states in favor of the transitions that the process can refuse to execute. However, Heymann (1991) shows that the trajectory model is more detailed than the failures model, in that the trajectory model can distinguish between nondeterministic processes that are not distinguished by the failures model. The key to this lies in the expression "the *environment*, offered them by means of the synchronous composition". The main process composition operator of the trajectory model is the *prioritized synchronous composition* (PSC).

The trajectory model models a discrete event process by its set of *trajectories*, elements of the set of observations $(2^{\Sigma} \times \Sigma)^* \times 2^{\Sigma}$. A trajectory is thus an ordered set of ordered pairs of a set of events and an event, followed at the end by a final set of events. The sets of events are called *refusals*. Note that these are *not* refusal sets as described above. Rather, a refusal is the set of all events the process can refuse to engage in at that point. See Example 2.2.

The prioritized synchronous composition of two processes $P$ and $Q$ with equal alphabets $\Sigma$, is by Heymann (1991) defined as $P_A \|_B Q$, where the priority sets $A$ and $B$ are both subsets of $\Sigma$. This partitions $\Sigma$ into the following four disjoint subsets. See also Balemi (1992) who relaxes the constraint of equal alphabets.

1. $A \bigcap B$ – the *strict synchronization* events. Either both processes execute these events or none.

2. $\Sigma - (A \bigcup B)$ – the *broadcast* synchronization events. Any of the two processes can execute such an event autonomously. If the other process can participate, it will.

3. $A - B$ – the *priority events* of $P$. Can occur if and only if $P$ participates. If the other process can participate, it will.

**Example 2.2 Trajectories and Prioritized Synchronization**

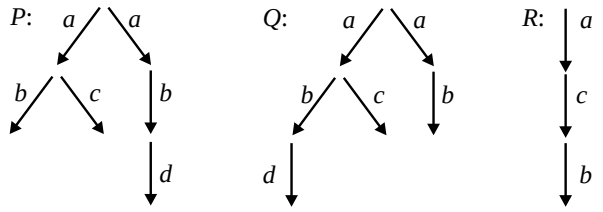



**Figure 2.9:** Three discrete event processes, $P$, $Q$ and $R$ with equal alphabets, $\Sigma = \{a, b, c, d\}$.

The trajectories of maximal length and with maximal refusals, describing the processes of Figure 2.9 are given in (2.67).

$$\begin{aligned}
\text{traj}(P) &= \left\{ \begin{array}{l} \langle \{b,c,d\}, a, \{a,d\}, c, \Sigma \rangle, \langle \{b,c,d\}, a, \{a,d\}, b, \Sigma \rangle, \\ \langle \{b,c,d\}, a, \{a,c,d\}, b, \{a,b,c\}, d, \Sigma \rangle \end{array} \right\} \\
\text{traj}(Q) &= \left\{ \begin{array}{l} \langle \{b,c,d\}, a, \{a,d\}, c, \Sigma \rangle, \langle \{b,c,d\}, a, \{a,c,d\}, b, \Sigma \rangle, \\ \langle \{b,c,d\}, a, \{a,d\}, b, \{a,b,c\}, d, \Sigma \rangle \end{array} \right\} \\
\text{traj}(R) &= \{ \langle \{b,c,d\}, a, \{a,b,d\}, c \{a,c,d\}, b, \Sigma \rangle \}
\end{aligned} \tag{2.67}$$

That these are the maximal trajectories, means that we do not list their prefixes, even though these should formally be included. That we only list the maximal refusals means to say that all conceivable trajectories with subsets of the refusals should formally also be included.

The prioritized synchronous composition with $A = \{a, b, d\}$ and $B = \{a, b, c\}$ of $P_A \| _B R$ and $Q_A \| _B R$ are shown in Figure 2.10.

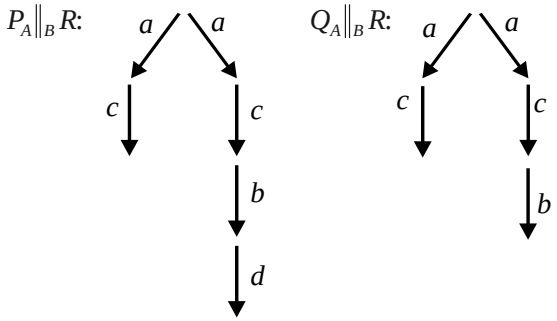



**Figure 2.10:** The processes resulting from the prioritized synchronous composition of the processes shown in Figure 2.9.

Example continued on next page

**Example 2.2 continued**

The models presented in this example are taken from Heymann (1991), Example 9.1. The interesting thing about them is that they are used by Heymann (1991) to show that the failures model cannot necessarily distinguish between processes that behave differently in concurrent composition with another process; not under prioritized synchronous composition, that is. The failures describing $P$ and $Q$ of Figure 2.9 are identical, yet when composing these processes with $R$ under the prioritized synchronous composition, the results are different, even as described by the failures model; the failures of $P_A\|_B R$ are different from the failures of $Q_A\|_B R$. This comparison may seem unfair to the failures model, since the prioritized synchronous composition is not included in the theory presented by Hoare (1985). Under full synchronous composition, the processes $P\| R$ and $Q\| R$ are identical.

■

4. $B - A$ – the *priority events* of $Q$. Can occur if and only if $Q$ participates. If the other process can participate, it will.

Naturally, if $A = \Sigma_P$ and $B = \Sigma_Q$, then the PSC is equivalent to the synchronous composition. However, if the priority sets are $A = B = \emptyset$, then the PSC is not equivalent to interleaving as defined in Definition 2.46. The phrase "If the other process can participate, it will" makes sure of this. Interleaving was defined as the total asynchronous execution of two processes. No mutual execution of any event is allowed. The PSC, as well as interleaving in the failures model, see Example 2.3, states that if both processes are in a state to execute a common event, then both will do so simultaneously, and no distinction can be made of which process actually executed (or initiated) the event. This is not so surprising, since neither the trajectory model nor the failures model uniquely distinguish the states. Several states may have the same refusals or refusal sets.

### 2.5.3 Automata Equivalence

The question of equivalence between DEPs is a delicate one. Hopcroft (1979) for instance, equates two processes that have the same language. Hoare (1985) equates two processes that have the same language and the same refusal sets. Heymann (1991), naturally, equates two processes that have the same trajectory model, while we in this work equate processes that have the same structural appearance, that are subprocesses of each other. Milner (1989) gives several definitions of automata equivalence.

Why do we need to discuss the aspect of DEP equivalence, at all? The supervisory control theory assumes the existence of a plant $P$ to be controlled, and a specification *Sp*, say, that defines the desired behavior of $P$. Both of these may be regarded as given as DEPs. Then we are to find a third DEP $S$, the supervisor, such that the composition of plant and supervisor equals the specification. This places some constraints on *Sp*, but we will disregard that for now, as we will disregard under what operation $P$ and $S$ are composed. However, the question of what we mean by process equivalence, comes into play here. If all of the processes are deterministic, then we can say that the composition of $P$ and $S$ are equal to *Sp*, if they generate the same language.

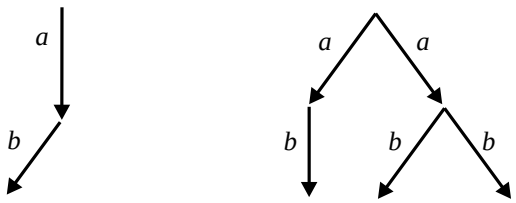



**Figure 2.11:** Two DEPs that are not distinguished by their externally visible behavior.

Equating two processes by they having equal languages is obviously not strong enough when we are considering nondeterministic automata. The failures model would equate the processes if they have the same language and the same failures, while the trajectory model equates them if their trajectories are the same. We have chosen to equate two processes when they are subprocesses of each other. The main reason for this is that, to us, the states hold significant information, determining the context under which a sequence of events was executed. Thus, we equate two transition machines if they have identical structural appearance.

A comment on such a definition of automata equivalence is given by Milner (1989) (italics at the end have been added for emphasis.)

> *... this is* too *strong, since there are cases of apparent nondeterminism which are not real. We shall want to equate the agents whose trees are as follows:* [see Figure 2.11] *since the extra branching on the right cannot be distinguished by any conceivable experiment,* if the experimenter is only able to detect which actions can occur at any stage.
>
> Milner (1989), p. 87

Furthermore, on page 208, Milner (1989) says "bisimilarity - or its associated congruence - the appears to be the strongest equivalence *based upon observable actions* that one can reasonable demand." (Italics added for emphasis); and Hoare (1985) p. 29, "Two objects which behave the same up to every moment in time have the same behavior, i.e. they are the same process". We can note that both Hoare (1985) and Milner (1989) (more or less) implicitly define *how* an automaton is to be experienced. From their point of view it is the *externally visible* behavior that is important. Again, Milner (1989), page 84, "... we only wish to distinguish between two agents $P$ and $Q$ if the distinction can be detected by an external agent interacting with them". Thus, the question of process equivalence gets down to what one means by the behavior of the process. One can get very philosophical about these aspects, but we will try to refrain from that here.

In the approaches of Hoare (1985), Milner (1989) and Heymann (1991) automata are distinguished by their externally visible behavior. This has been emphasized in the citations above. The way an automaton is experienced is through the events it executes, and the events it may refuse to execute. The specific states are not deemed important. Thus, a refusal-trace "does not uniquely determine the state" (Shayman (1994), Remark 1). We distinguish automata by its *internally* modeled behavior.

As pointed out by Shlaer (1992), describing a DEP from its externally visible behavior, and from its internally modeled behavior, results in two very different views of the process.

Modeling the externally observable behavior regards the modeled system as a "black box", whose inner workings are not known. States are formed in accordance with what can be observed from the outside, and information conveyed by the states or actions performed in the states have to be omitted. By contrast, our approach, as well as the approach of Shlaer (1992), seek to explain the details of the behavior from inside the system. Every state has a purpose and a context. The full meaning of a state is not necessarily conveyed by the string of events reaching that state, but also by the states preceding, and perhaps even following that state. Shlaer (1992), gives models of a microwave oven both from the externally visible behavior view-point, and from the internal perspective. Differences between these two approaches are also described for a mixing tank.

To supervise any system, or at least to calculate a supervisor for any system, we need an adequate model representing that system. Since any object encompasses an infinite number of properties, of which any model can only hope to catch a limited amount, any model is an abstraction of the actual object. Of importance then, is that this abstraction captures the specific properties relevant to the task at hand, and as little as possible of those properties of lesser or no interest. For nondeterministic processes we know that the language model does not capture all relevant aspects of the DEP. As has been discussed above, for our purposes, it is not enough to distinguish two processes by their externally visible behavior. We need to know the actions leading up to specific states, even though these actions may be equivalent from an external point of view. Specifically, we need to know which individual product has undergone what sequence of events. For this purpose, we may need to distinguish between states that the language, failures, trajectories or the Milner-equivalences do not care to distinguish. Thus, we may have to distinguish between the two automata of Figure 2.11, since the states represent different contexts under which the $a$ and $ab$ strings were generated. See also, Example 2.3.

The trajectory model is used by Shayman (1994), who use the prioritized synchronous composition to model driven events. The supervisory control theory is extended to this case, including non-deterministic plants. See Section 3.4.

### 2.5.4 Event Generation

Formally, there is no difference between viewing an automaton as a generator or an acceptor of events, as pointed out by Ramadge (1987). However, an *acceptor* is a device which is driven by events generated by some external source, its *environment*. The acceptor merely informs the environment of whether the generated string was "recognized" or not. This is the interpretation of an automaton given by Hopcroft (1979). We can view such a device as a box with a number of labeled buttons on it; one for each event of the alphabet, a reset button and a button with a question mark on it. In each state of the automaton, the buttons corresponding to the events of the ready set are lit. When pressing such a button, we generate an event that forces the automaton to transit to a new state. In this state, a new set of buttons may light up. When we are done generating events, we push the question mark button, thus requesting the information of whether the generated string was recognized by the device, or not. If the automaton is then in a marked state, a green lamp is lit, otherwise a red lamp. Now we have the possibility to generate new events from the present state, or reset the automaton and play again. What is important to note here, is that it is we, as the environment, that generates the events, and thus we

also determine when a string is ended. If the green lamp lights up, that string is in the accepted language of the automaton.

A generator on the other hand, has no buttons to push. Only small lamps labeled with the events. When an event is generated, the lamp corresponding to that event is lit, so that we can observe its generation. This time, it is the automaton that determines when it is finished generating a sequence of events. Since we do not know the amount of time the automaton spends between event-generation, it seems reasonable that we regard all strings as "finished", at any time. This would mean that the language generated is prefix closed, whereas, for the acceptor, the language accepted is not necessarily prefix closed. This is also the interpretation given by Ramadge (1987); the plant to be controlled generates all the events, and so its language is prefix closed. Hopcroft (1979), on the other hand, regards an automation as an acceptor, and so the language represented by the automaton is the marked language. In this context, it is also strange to regard a generator as having marked states. In this work we do not, Ramadge (1987) do.

In the work of Hoare (1985), marking is not regarded. In fact, the distinction between generated and accepted events is entirely disregarded. Hoare (1985) claims that "... there is no need to make a distinction between events which are initiated by the object [...] and those which are initiated by some agent outside the object." (page 24)

However, the distinction between *generated* and *accepted* event is formally introduced by Milner (1989). For every event $a$ there exists a complement event, denoted $\overline{a}$. The $a$ event is the generated event and its complement is the accepted event. In practice, when two processes are brought together to evolve concurrently, some events are generated by one of the processes while some events are generated by the other. Thus, when they are to synchronize, it makes sense to distinguish between the generated and the accepted event.

Every process $P$ defines its alphabet as $\Sigma_P = \Sigma'_P \bigcup \overline{\Sigma}_P$, where $\Sigma'_P$ are the events generated by $P$, and $\overline{\Sigma}_P$ are the events accepted by $P$. Milner's composition operator is, with the exception of treatment of the *silent* (null) event $\tau$, equal to $P \|_\Sigma Q$ where the generated events of $P$ are synchronized with the accepted events of $Q$, and vice versa.

## 2.6 Chapter Summary

In this chapter we have described *finite transition machines* that will be used to model discrete event processes. A global universe of which all transition machines are elements have been defined to bring meaning to a number of operators and relations between transition machines. A number of properties have been defined and shown. The main points of this chapter are the following

1. When a process $S$ refines a deterministic process $P$, then the synchronous composition $P \| S$ is equivalent to $S$. Equivalence is in this case defined as the strong equivalence given by the structur of the processes being equal.

2. When $S$ refines $P$ it holds that $L(S) \subseteq L(P)$.

3. Any subprocess refines its superprocess, and the subprocess relation is a partial ordering. Therefore, when $S$ refines $P$, any subprocess $S'$ of $S$ refines $P$, so that $P \| S'$ is equivalent to $S'$ when $P$ is deterministic.

**Example 2.3 Types of Interleaving**

The following example is adapted from Hoare (1985), Example X1 of Section 3.6.

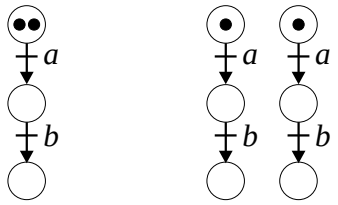



**Figure 2.12:** Two Petri nets. The right net consists of two identical interleaved nets, while the left one consists of one net with two tokens.

The difference between the failures model and the transition machine model is best shown by the two Petri nets of Figure 2.12. The left Petri net consists of one set of place/transitions, with two tokens. In such a Petri net, the tokens have no identity (unless they are colored), and so firing one is identical to firing the other. The reachability graph for this net is shown to the left in Figure 2.13. This is, in practice the failures model resulting from interleaving two identical processes like the ones given in Figure 2.12. For a given marking, we cannot determine which token has executed what sequence of events. This is identical to the failures model description. The failures model does not uniquely determine the state of a nondeterministic state-machine. The refusal sets after one of the tokens has initially fired the $a$ transition is no different from the refusal set after the other token has initially fired the same transition. Therefore, the failures model does not distinguish between states that it might be necessary to distinguish between in practice. If the tokens represents products to be produced, it may very well be critical to be able to determine *which* product has executed which sequence of events.

The right Petri net of Figure 2.12 consists of two identical sets of place/transitions, with one token each. In such a net, the tokens are individual and have their own unique identity. This net is in fact the transformation into ordinary Petri nets of the net to the left of Figure 2.12, had the tokens there been colored. The reachability graph of the right Petri net of Figure 2.12 is shown to the right of Figure 2.13. This is the transition machine resulting from interleaving two identical transition machines like the ones to the right in Figure 2.12. As can be seen, this transition machine distinguishes between each token. The state $(1,0,0,0,0,1)$ is different from the state $(0,0,1,1,0,0)$, even though they are both reached by the string $ab$. It may be crucial for the control system to know that in state $(0,0,1,1,0,0)$ it is product number one that has executed the string $ab$.

Example continued on next page

**Example 2.3 continued**

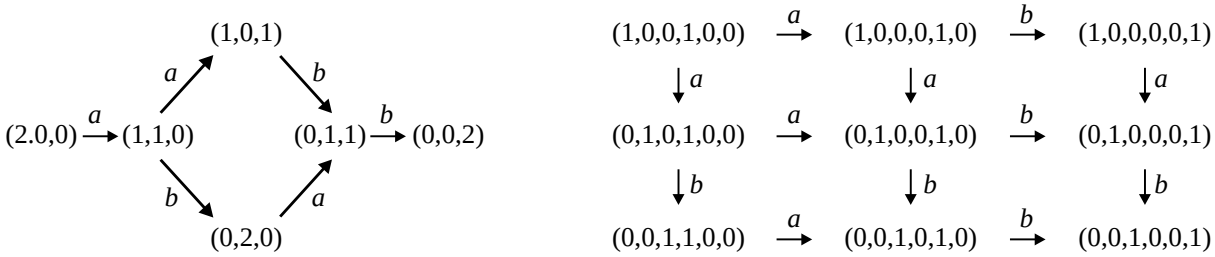



**Figure 2.13:** The reachability graphs of the two nets given in Figure 2.12. The left one does not maintain the uniqueness of the tokens, while the right one does.

Interleaving two identical transition machines as the ones given to the right in Figure 2.12 in their failures model representation, would result in the left process of Figure 2.13. The thing is that the failures model does not distinguish between states reached by the same string, if the refusal sets of those states are equal. This also holds for the trajectory model, as noted by Shayman (1994) in Remark 1. Of course, it is not always the case that such states must be distinguished. Overkamp (1995) and Shayman (1994) do not seem to need this. In such cases, the failures model, as well as the trajectory model, is applicable and can be used as reduced state models.

Finally, we can note that the left transition machine of Figure 2.13 is deterministic, while the right one is not.

■

4. For any two processes $P$ and $S$, the synchronous composition $P\|S$ refines $P$, whether $P$ is deterministic or not.

This has the consequence that given a deterministic process $P$ and a process $S$, not necessarily deterministic, a subprocess $S'$ of $P\|S$ refines $P$, so that $P\|S'$ is equivalent to $S'$ and is thus a subprocess of $P\|S$.

The importance of this observation stems from the interpretation of $P$ as a deterministic plant to be controlled, and $S$ as a specification process, with $P\|S$ as the specified achievable behavior. Then, $S'$ can be interpreted as the controlling entity, the supervisor, influencing $P$ so that a part of the specified behavior is accomplished. This part can be regarded as the *controllable* part of the specification, and in the next chapter will be given conditions for such a part to exist.

The definitions and results presented in this chapter are essentially well-known, though the notion of refinement is extended to non-deterministic processes. The observation that the synchronous composition of $P\|S$ is equal to $S$, whenever $P$ is deterministic and $S$ refines $P$, Theorem 2.48, is an important result that seems to be new. Particularily since we show that the synchronous composition of $P$ with *any* process refines $P$. This fact is implicitly used in Kumar (1991), among others, where $P\|S$ is used as a supervisor "candidate" to be made complete. The definitions of operations on subprocesses, Definition 2.50, seem to be novel. In the literature the state-spaces are considered to be disjoint and the event-sets equal; see Eilenberg (1974) and Hopcroft (1979), for instance.

# Chapter 3

# Supervisory Control

In this chapter we will show how the supervisory control theory (SCT) can be used in the more general setting of nondeterministic discrete event processes. We will give formal proofs for existence and uniqueness of a number of differently specified supervisors. First a brief background regarding some of the various formulations of the SCT found in the literature, is given . Naturally, all of these are based on the pioneering work of Ramadge (1987) and Wonham (1987). Then, the input/output formulation of Balemi (1992) is described in some detail, followed by our approach to nondeterministic supervisory control.

## 3.1 Introduction

The SCT views the DEP to be controlled, that is, the *plant*, as a *generator* of events. All events arise within the system as a consequence of some action taking place within the plant, spontaneously and asynchronously. The controlling entity, the *supervisor*, has to follow the generated events, while having the ability to disable the generation of some events. This ability to disable the generation of plant events is in Ramadge (1987) exercised by the use of a *feedback map*, determining for each supervisor state which plant events to disable.

However, in Kumar (1991) the full synchronous composition of plant and supervisor as a model of the controlled system, is introduced . Here the plant and the supervisor operate in full synchrony and only those events simultaneously defined by the plant and the supervisor can be generated. Thus, the ability to disable events is applied by not defining those events in the relevant supervisor state. This latter approach significantly simplifies the analysis of the supervisor synthesis problem.

The event set of the plant is partitioned into two disjoint event sets, the *controllable* and the *uncontrollable* events. The controllable events are subject to influence by the supervisor, whereas the uncontrollable events are not. Despite the formal definition of the full synchronous composition, the plant can generate the uncontrollable events as it pleases. Thus, for a supervisor $S$ to guarantee that the controlled closed-loop system behaves within the given specification, $S$ must at no time try to disable any of the uncontrollable events. Such a supervisor is said to be *complete* with respect to the plant and the uncontrollable events.

In Ramadge (1987) the behavior of a plant is defined by the sequence of events, the *language*, it can generate. Thus, for the controlled system's behavior a specification

language can be given, and it is shown that for a complete supervisor to exist, this language must be *controllable*. That is, for any string of the specification language that can be followed by an uncontrollable event in the plant language, that uncontrollable event must also be allowed within the specification language. It can be shown, see the remark to Lemma 3.3 below, that for a deterministic supervisor completeness is equivalent to controllability of its language.

Other interpretations of the generation of events in the interaction between the plant and the supervisor have been investigated. For instance, the input/output formulation by Balemi (1992), which is described in Section 3.3. Heymann (1991) introduces the prioritized synchronous composition to model the interaction between supervisor and plant, primarily to include *driven* events by which the supervisor can force actions to occur within the plant. This plant/supervisor interaction model with driven events is also used by Shayman (1994) and Kumar (1994). Balemi (1992) uses the prioritized synchronous composition to model to model the interaction between a *controller* and the plant, introducing *command* and *response* events.

## 3.2 The Basic Supervisory Control Theory

A supervisor for a deterministic plant $P = (Q_P, \Sigma_P, i_P, Q_P, E_P)$ is a deterministic process $S = (Q_S, \Sigma_S, i_S, M_S, E_S)$ that operates in full synchronization with the plant, thereby influencing the plant's behavior. Note that the initial states are given as singular $i_P$ and $i_S$, respectively, emphasizing that we are dealing with deterministic processes in this section. Note also that the plant is considered to be *unmarked*, that is $M_P = Q_P$, and $L_m(P) = L(P)$. Ramadge (1987) allows the plant to be marked as well, interpreting marking as a modeling option to signify, for instance, completion of specific tasks. But marking, as we interpret it, is a type of specification; states that the closed-loop system of plant and supervisor has to be able to reach are marked. Other approaches, such as Overkamp (1994) and Giua (1991) have considered non-marked supervisors.

We will always assume that the alphabet of the supervisor is equal to the alphabet of the plant. That is, $\Sigma_S = \Sigma_P$. This assumption means that the supervisor specifies the *whole* controlled behavior. An event of $\Sigma_S$ that does not appear in any string of $L(S)$ is forever disabled, and only events present in some string of $L(S)$ can ever be enabled. This assumption also significantly simplifies all notations and proofs, and is commonly accepted. See for instance Ramadge (1987), Kumar (1991), Heymann (1991) and others.

Given a plant $P$ and a specification language $K$ defining the closed-loop system's behavior, the supervisor synthesis problem is a matter of finding a supervisor $S$, such that $L(P \| S) = K$. Note that this implicitly requires $K$ to be prefix closed, since $L(P \| S)$ is prefix closed by definition.

However, since the event set of $P$ is partitioned into the controllable and the uncontrollable events, $\Sigma_c$ and $\Sigma_u$, respectively, the supervisor must ensure the specification, without ever trying to disable an uncontrollable event. This is the notion of *completeness* defined by Ramadge (1987). A supervisor $S$ is complete, with respect to a plant $P$ with an uncontrollable event set $\Sigma_u$, if and only if in the closed-loop system of $P \| S$, no uncontrollable events that may be generated by $P$ are ever disabled by $S$. Formally, we have the following definition.

**Definition 3.1 Completeness**

A supervisor $S$ is said to be *complete* with respect to a plant $P$ and a set of uncontrollable events $\Sigma_u$, if and only if

$$\forall s \in L(P \| S)\ \mathrm{out}(\delta(P,s)) \bigcap \Sigma_u \ \subseteq \mathrm{out}(\delta(S,s)) \tag{3.1}$$

**Remark.** Since the ready sets of the states of the composed system $P \| S$ are the intersection between the corresponding ready sets of the individual processes, a supervisor $S$ is complete if the ready set for a state $q$ reached by a string $s$ arising in the closed-loop system, contains the uncontrollable events of the ready set of the state $p$ reached by $s$ in the plant $P$. Thus, if (3.1) holds, no uncontrollable events following strings allowed in the controlled system are ever disabled.

Note that $S$ and $P$ being deterministic is essential here. Otherwise, neither $\mathrm{out}(\delta(P,s))$ nor $\mathrm{out}(\delta(S,s))$ would be well defined, since, in general, $\delta(\cdot,\cdot)$ is not a singleton. However, when $S$ and $P$ are deterministic, $\delta(P,s)$ and $\delta(S,s)$ are both single states. Note also that we have assumed that the supervisor exercises control by executing in full synchrony with the plant. Thus, events are generated if and only if both the plant and the supervisor participate. Disabling events is therefore a matter of the supervisor not agreeing to participate in the event. Naturally, all uncontrollable events that are possible under supervision must be agreed upon by the supervisor for it to properly control the plant. This is just what completeness says.

There are other ways to express supervisor completeness. For instance, Balemi (1992) uses the prioritized synchronization operator to express supervisor completeness. Then, a supervisor $S$ is complete with respect to a plant $P$ with uncontrollable events $\Sigma_u$, if the full synchronous composition $P \| S$ is equal to the prioritized synchronous composition $P_\Sigma \|_{\Sigma_u} S$.

Completeness of a supervisor $S$ only concerns the states of $Q_S$ that are accessible in the closed-loop system. Only states reached by strings in $P \| S$ are considered by Definition 3.1. Just as there may exist regions of $Q_P$ that we do not want the closed-loop system to visit, so may there exist parts of the supervisor $S$ that are not accessible in the closed-loop system. Such parts of $S$ are, in practice, of no interest to us. With the assumption that the plant generates all events, these parts of $S$ will never be entered by the closed-loop system. Therefore, it is perfectly reasonable to assume that $L(S) \subseteq L(P)$, that is, the supervisor only *restricts* the behavior of the plant. Then, $L(P \| S) = L(P) \bigcap L(S) = L(S)$, so that we could replace $L(P \| S)$ with $L(S)$ in the definition above. Note that in the literature, this replacement is often done without any comments. □

In Lemma 2.44 was shown that when $\Sigma_S \subseteq \Sigma_P$ then $L(P) \bigcap L(S)\Sigma_{P-S} \subseteq L(P \| S)$. It was also mentioned in the remark that the events of $\Sigma_{P-S}$ could be regarded as *uncontrollable* to $S$. Assume now that $\Sigma_S = \Sigma_P = \Sigma_c \bigcup \Sigma_u$. When $S$ cannot control the generation of the events of $\Sigma_u$, these "behave" in the same way as the events of $\Sigma_{P-S}$ of Lemma 2.44 even though $\Sigma_u$ is a subset of $\Sigma_S$. Then the following inclusion holds

$$L(P) \bigcap L(S)\Sigma_u \subseteq L(P \| S). \tag{3.2}$$

Formally, though, by Corollary 2.45 we have that the language of the synchronous com-

position of two processes with equal alphabets is

$$L(P\| S) = L(P) \bigcap L(S). \tag{3.3}$$

For $S$ to be able to control $P$ so that the specification language is guaranteed, both the expressions of (3.2) and (3.3) must hold. That is, we must have that

$$L(P) \bigcap L(S)\Sigma_u \subseteq L(P) \bigcap L(S). \tag{3.4}$$

This expression is obviously (see Lemma A.1) equivalent to

$$L(P) \bigcap L(S)\Sigma_u \subseteq L(S). \tag{3.5}$$

When this holds, $L(S)$ is said to be *controllable* with respect to the plant. This is also defined by Ramadge (1987). For an arbitrary prefix closed language, $K$, the following definition of controllability can be made.

**Definition 3.2 Controllability**

For a plant $P$ with language $L(P)$ and a set of uncontrollable events $\Sigma_u$, a prefix closed specification language $K$ is controllable with respect to $P$ if and only if for any string $s \in K$ and for any uncontrollable event $\sigma_u \in \Sigma_u$ such that $s\sigma_u \in L(P)$ we have that $s\sigma_u \in K$. That is

$$K \text{ controllable} \Leftrightarrow K\Sigma_u \bigcap L(P) \subseteq K \tag{3.6}$$

**Remark.** Intuitively, controllability means that for a string $s$ of $K$ that exists in $L(P)$ and can be followed by an uncontrollable event in $L(P)$, this uncontrollable event must also be able to follow $s$ in $K$. This means that, if the plant uncontrollably takes the system somewhere, the specification has to be able to "follow". Thus, controllability ensures that it is possible to control the plant so as to stay within the language $K$. □

Note that controllability is a property of the *language* defined by a transition machine, whereas completeness is a property pertaining to the *structure* of the transition machine. These two notions are obviously closely related, in fact, for deterministic automata they are equivalent.

**Lemma 3.3** A supervisor $S$ is complete with respect to a plant $P$ and a set of uncontrollable events $\Sigma_u \subseteq \Sigma_P$ if and only if $L(S)$ is controllable.

**Proof.** By the definitions of active events, ready sets, controllability and completeness the following expressions are all equivalent.

$$\begin{aligned} & L(S)\Sigma_u \bigcap L(P) \subseteq L(S) \\ \forall s \in L(P\| S)\ & \gamma(L(P), s) \bigcap \Sigma_u \subseteq \gamma(L(S), s) \\ \forall s \in L(P\| S)\ & \text{out}(\delta(P, s)) \bigcap \Sigma_u \subseteq \text{out}(\delta(S, s)) \end{aligned} \tag{3.7}$$

Thus, controllability of $L(S)$ (at the top) is equivalent to completeness of $S$ (at the bottom). ■

**Remark.** Again we note that it is essential that we deal with deterministic processes. Otherwise, $\gamma(L(P), s)$ and $\gamma(L(S), s)$ are, respectively, not necessarily equal to $\text{out}(\delta(P, s))$ and $\text{out}(\delta(S, s))$.

That completeness of $S$ is equivalent to controllability of $L(S)$, is also proved by Balemi (1992), though the term *controllability* is never defined. The proof merely states that when $S$ is complete, then (3.5) is satisfied, and vice versa. Note also the simplicity of the proof above relative to the involved proof of Balemi (1992), who uses *language projection* to arrive at the desired result. □

Naturally, $L(P)$ is always controllable. Because of this, the following lemma can be proved.

**Lemma 3.4** For a supervisor $S$, its language $L(S)$ is controllable with respect to a plant $P$ and a set of uncontrollable events $\Sigma_u \subseteq \Sigma_P$, if and only if $L(P \| S)$ is controllable.

**Proof.** Since $L(P)$ is always controllable the following expressions are all equivalent.

$$\begin{aligned} L(P \| S)\Sigma_u \bigcap L(P) &\subseteq L(P \| S) \\ \Big[L(P) \bigcap L(S)\Big]\Sigma_u \bigcap L(P) &\subseteq L(P) \bigcap L(S) \\ L(P)\Sigma_u \bigcap L(S)\Sigma_u \bigcap L(P) &\subseteq L(P) \bigcap L(S) \\ L(S)\Sigma_u \bigcap L(P) &\subseteq L(S) \end{aligned} \tag{3.8}$$

Therefore, $L(S)$ is controllable, if and only if $L(P \| S)$ is controllable. ■

**Remark.** Thus it follows that a supervisor $S$ is complete, if and only if $L(P \| S)$ is controllable.

This proposition is also given by Kumar (1991), Lemma 2.7. □

Next, we give necessary and sufficient conditions for the existence of a complete supervisor $S$, given a plant $P$ and a specification language $K$.

**Lemma 3.5** Let $K$ be a prefix closed specification language for a plant $P$. Then there exists a complete supervisor $S$ such that $L(P \| S) = K$, if and only if $K \subseteq L(P)$ and $K$ is controllable.

**Proof.** ($\Rightarrow$) Since we assume $\Sigma_P = \Sigma_S$ we have that $L(P \| S) = L(P) \bigcap L(S) = K \subseteq L(P)$.

Since $S$ is complete, we know by Lemma 3.4 that $L(P \| S)$ is controllable.

($\Leftarrow$) We have that $K \subseteq L(P) \Leftrightarrow L(P) \bigcap K = K$. Choose $S$ such that $L(S) = K$. Since $K$ is prefix closed (and regular) this can always be achieved. Then $L(P \| S) = L(P) \bigcap L(S) = L(S) = K$. Thus, $L(S)$ is controllable, and by Lemma 3.3 $S$ is complete. ■

**Remark.** This proof is also given by Ramadge (1987) as a part of Proposition 5.1.

Formally, $L(S) = K$ is only *one* possible choice. Any supervisor with a language such that $L(P) \bigcap L(S) = K$ would be suitable. However, by the remark to the definition of completeness, Definition 3.1, $L(S) \subseteq L(P)$ is a perfectly reasonable assumption. In that case, only supervisors such that $L(S) = K$ are possible. This assumption also means that $L(P \| S) = L(P) \bigcap L(S) = L(S)$, and thus, the supervisor is a model of the plant under supervision. □

In a manner similar to (3.2), Lemma 2.44 also shows that $L(P) \bigcap L(S)\Sigma_u^* \subseteq L(P \| S)$. Obviously, it is always the case that

$$L(P) \bigcap L(S) \subseteq L(P) \bigcap L(S)\Sigma_u^* \tag{3.9}$$

since

$$L(S) \subseteq L(S)\Sigma_u^*, \tag{3.10}$$

so that $L(S)$ controllable is equivalent to

$$L(P) \bigcap L(S) = L(P) \bigcap L(S)\Sigma_u^*. \tag{3.11}$$

By, (3.11), controllability can also be expressed by $L(P) \bigcap L(S)\Sigma_u^* \subseteq L(S)$. This is also noted and proved by Brandt (1990), Lemma 1, as well as, though without proof, by Kumar (1991) in Corollary 3.3.

For *any* language $K \subseteq L(P)$, there does exist a supervisor such that $L(P \| S) = K$, irrespective of whether $K$ is controllable or not. However, though formally the language of the closed-loop system is still equal to $K$, in practice this will not be the case when $K$ is non-controllable. This is a fact arising from the usage of the synchronous composition operator to model the interaction of plant and supervisor. The aspect of uncontrollable events is not included in the definition of the synchronous operator, as shown above. Thus, the plant can generate the uncontrollable events as it pleases. Thus, the synchronous composition $P \| S$ is not an adequate model of the closed-loop system. In fact, Balemi (1992) uses the prioritized synchronous composition to model the closed-loop system, requiring that $P \| S = P_\Sigma \|_{\Sigma_u} S$. This expression is used by Balemi (1992) to *define* both controllability and completeness.

Not all specification languages $K$ such that $K \subseteq L(P)$ are controllable. Then the *controllable sublanguage* of $K$ has to be considered. The set of all controllable sublanguages of a specification language $K$ is known to be closed under language union. Therefore there exists a *supremal controllable sublanguage*, denoted $K^\uparrow$. For a given specification language $K$, we always have $K^\uparrow \subseteq K$.

Now we know that a supervisor $S$ such that $L(P \| S) = K^\uparrow$ will be complete and allows the largest achievable behavior within the specification language $K$. The supervisor $S$ is then said to be *minimally restrictive*. However, we also want the supervisor to uphold the marked specification language as well as possible. The largest achievable marked behavior within the marked specification language is of course $K_m \bigcap K^\uparrow$. This is the largest marked behavior that is (in the words of Ramadge (1987)) "consistent with the controlled behavior". But there is more to this. An aspect of great importance is that we always want the closed-loop system to be able to reach a marked state. We can look at the marked states as states wherein the system is allowed to rest. Typically marked states denote the completion of some sub-task of the system, after which a new task proceeds.

Thus, we do not want to allow the closed-loop system to indefinitely transit between some unmarked states, even though this behavior may be allowed by the controllability property. We want the closed-loop system to be nonblocking, see Definition 2.22. Note that with a deterministic transition machine, there is one, and only one, state reached by each string. Thus, a deterministic transition machine has a nonblocking language if and only if every accessible state is also coaccessible.

By the remark to Lemma 3.5 we can always assume that the supervisor is a model of the closed-loop system under supervision, that is, $L(S) \subseteq L(P)$. Then, of course, $L_m(S) \subseteq L(P)$. Because of this when the supervisor is nonblocking, the closed-loop system will also be. See also Corollary 3.30. The following theorem gives necessary and sufficient conditions for the existence of a nonblocking and complete, deterministic, marked supervisor.

**Theorem 3.6** Let $K_m \subseteq K$ be two specification languages for a plant $P$ with $K$ prefix closed. Then there exists a nonblocking and complete, supervisor $S$ such that $L_m(P\|S) = K_m$, $L(P\|S) = K$ and $\overline{L_m(P\|S)} = L(P\|S)$ if and only if $K_m \subseteq \overline{K_m} = K \subseteq L(P)$, and $K$ controllable.

**Proof.** ($\Rightarrow$) When $S$ is complete $L(P\|S) = K$ is controllable. Then $L(P\|S) = L(P) \bigcap L(S) = K$ so that $K \subseteq L(P)$. The marked language $L_m(P\|S) = L(P) \bigcap L_m(S) = K_m \subseteq L(P) \bigcap L(S) = K$, since by definition $L_m(S) \subseteq L(S)$. Furthermore, we have that $\overline{L_m(P\|S)} = \overline{K_m} = L(P\|S) = K$.

($\Leftarrow$) Choose $S$ such that $L(S) = K$. When $K$ is controllable, $S$ is complete. Let $L_m(S) = K_m$, this can always be achieved since $K_m \subseteq K$. Then, since $K_m \subseteq K \subseteq L(P)$, $L(P\|S) = L(S)$ and $L_m(P\|S) = L_m(S)$. When $\overline{K_m} = K$ then $\overline{L_m(S)} = L(S)$, so that $S$ is nonblocking and $\overline{L_m(P\|S)} = L(P\|S)$. ∎

**Remark.** When $K = K^{\uparrow}$ we also know that the supervisor $S$ is minimally restrictive.

Similar proofs are given in Propositions 5.1 and 6.1 of Ramadge (1987) and Proposition 2.8 of Kumar (1991). We can note the simplicity of proving the existence of a complete and nonblocking supervisor, due to the use of the synchronous composition to model control of a discrete event process. This model of supervisor-plant interaction is also used by Kumar (1991), but the proof above is even simpler than the proof of Proposition 2.8 of Kumar (1991), even though nonblockingness is not treated in that work. □

The supervisory control problem now consists of finding the supremal controllable sublanguage, $K^{\uparrow}$, for given specification languages, $K_m$ and $K$. Once this language has been found, by the above, we know that we can find a supervisor $S$ such that, for the plant $P$, $L(P\|S) = K^{\uparrow}$ and $L_m(P\|S) = K_m \bigcap K^{\uparrow}$. If, in addition, $\overline{K_m \bigcap K^{\uparrow}} = K^{\uparrow}$, the resulting closed-loop system will be nonblocking.

### Summary

In this section we have given a brief overview of the original supervisory control theory as presented by Ramadge (1987) and Wonham (1987). The important proofs of those works have been given in the notation presented earlier. Where indicated, the proofs that are

presented are slight reformulations of the proofs given in Ramadge (1987), Kumar (1991) and Balemi (1992) to better suit the notation given in this thesis. The main difference between this chapter and the original work of Ramadge and Wonham lies in the use of the full synchronous composition to model the interaction between the supervisor and the plant. Originally a control map was used together with a supervisor automaton, see Ramadge (1987), which, for every supervisor state, specified the allowed events. In essence, with the synchronous composition the ready sets of the supervisor states model this control map. This has the added property, as has been shown, that the synchronous composition of the plant and the supervisor equals the supervisor itself, so that the supervisor is a model of the plant under control, the closed-loop system. This is not necessarily so in the model originally proposed by Ramadge and Wonham. The closed-loop system could, for instance, be trim without the supervisor automaton being trim, since it was this automaton together with the control map that specified the controlled behavior. Use of the synchronous composition to model the interaction between the plant and the supervisor is also described by Kumar (1991), though nonblockingness is not treated.

In the next section the supervisory control theory will be extended to another interpretation of the interaction between the supervisor and the plant, the input/output formalism of Balemi (1992), where the supervisor no longer is just a passive device following the plant and restricting the event generation, but also generates events of its own. This view lies closer to a control engineering point of view, a supervisor acting as a controller receives input from the plant and generates output to the plant.

## 3.3 The Input/Output Interpretation

In the original model proposed by Ramadge (1987) the plant generates *all* events, with the supervisor being a passive follower, dynamically restricting the choices of events to generate. However, from a control engineering point of view, it seems more natural to regard the plant as generating some kind of output in accordance with some input. Thus, it is natural to also view the supervisor as an active device, a *controller*, generating the input to the plant, the *process*, to be controlled.

It is observed by Balemi (1992) that in most real life systems events are seldom generated spontaneously, but only as *responses* to *commands*. For instance, a manufacturing device does not start its processing spontaneously, but only as a consequence of some start signal. Balemi (1992) equates the commands with the controllable events, and the responses with the uncontrollable events. For a control engineer this seems an intuitive interpretation of the controllable and uncontrollable events. We are free to determine when to enforce some control action, the command, but we cannot control its outcome, the response. The plant automaton can thus be interpreted as the (discrete event) transfer function between the input commands and the output responses. The differences between the original Ramadge/Wonham interpretation and the input/output interpretation are illustrated in Figure 3.1. The original Ramadge/Wonham approach to the left in Figure 3.1 is referred to as the *asymmetric* feedback loop, by Balemi (1992). The plant/supervisor connection resulting from the input/output interpretation, to the right in Figure 3.1, is referred to as the *symmetric* feedback loop by Balemi (1992).

Naturally, the supervisor must always be prepared to accept the responses generated

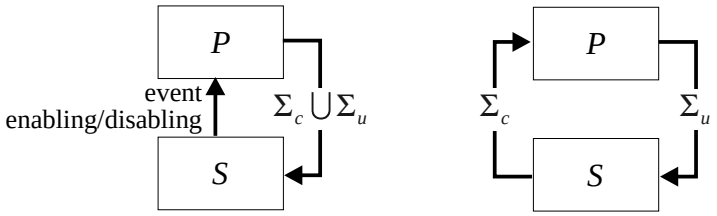



**Figure 3.1:** The original Ramadge/Wonham approach (left) and the input/output interpretation (right).

by the plant, otherwise they become "out of synch" with each other. Since the interaction between the plant and supervisor is modeled by full synchronous composition, this means that in each reachable closed-loop system-state the ready set of the supervisor must include all uncontrollable events that are in the ready set of the plant. In other words, the supervisor $S$ must be complete with respect to the plant $P$ and the uncontrollable events $\Sigma_u$, the responses.

Supervisor completeness is a necessary and sufficient requirement for the original Ramadge/Wonham interpretation of plant/supervisor interaction, since the plant is regarded as generating all events. However, for the input/output interpretation, supervisor completeness is not enough. It is a requirement that suffices for the acceptance of the responses by the supervisor, but the supervisor cannot be allowed to generate its commands haphazardly. The plant must be able to accept every command issued by the supervisor[1].

In Balemi (1992) the plant is then said to be *complete with respect to the supervisor*. However, since the plant is given and the supervisor chosen, this is actually an additional constraint on the behavior of the supervisor. It must only generate events which the plant can follow. Therefore, we will say that the supervisor must be *inverse complete* with respect to the plant and the controllable events.

### Definition 3.7 Inverse Completeness

A supervisor $S$ is said to be *inverse complete* with respect to a plant $P$ and a set of controllable events $\Sigma_c$, if and only if it only generates controllable events (commands) that $P$ can accept. That is

$$S \text{ inverse complete} \Leftrightarrow \forall s \in L(P \| S)\ out(\delta(S,s)) \bigcap \Sigma_c \subseteq out(\delta(P,s)). \tag{3.12}$$

**Remark.** Definition 3.7 is of course the dual to the definition of supervisor completeness, Definition 3.1, and as was the case with that definition, Definition 3.7 implicitly concerns only deterministic supervisor and plant. □

There also exists a dual to Definition 3.2 of language controllability. We say that a prefix closed language $K$ is inverse controllable with respect to a plant language $L(P)$ and a set of controllable events $\Sigma_c$ if a string $s$ of $K$ that exists in $L(P)$ and can be followed by a

[1] Note that this is very different from the *driven* events of Shayman (1994). It is an essential feature of the control design of Shayman (1994) that the plant is able to refuse a supervisor-initiated event.

controllable event in $K$, then $s$ can also be followed by that controllable event in $L(P)$. Formally we have the following definition.

**Definition 3.8 Inverse Controllability**

For a plant $P$ with language $L(P)$ and a set of controllable events $\Sigma_c$, a prefix closed language $K$ is *inverse controllable* with respect to $P$ if and only if for any string $s \in L(P)$ and for any controllable event $\sigma_c \in \Sigma_c$ such that $s\sigma_c \in K$ we have that $s\sigma_c \in L(P)$. That is

$$K \bigcap L(P)\Sigma_c \subseteq L(P). \tag{3.13}$$

**Remark.** This is the direct dual of Definition 3.2, with $K$ and $L(P)$ interchanged, and $\Sigma_c$ in place of $\Sigma_u$. □

Note that inverse controllability is a property of the *language* defined by a transition machine, while inverse completeness is a property of the transition machine. Just as was the case with controllability and completeness. Of course, the notions of inverse controllability and inverse completeness are closely related; so much that for deterministic transition machines inverse controllability and inverse completeness are equivalent.

**Lemma 3.9** A supervisor $S$ is inverse complete with respect to a plant $P$, with a set of controllable events $\Sigma_c$, if and only if $L(S)$ is inverse controllable.

**Proof.** By the definitions of active events, ready sets, inverse controllability and inverse completeness the following expressions are all equivalent.

$$\begin{aligned} & L(S) \bigcap L(P)\Sigma_c \subseteq L(P) \\ \forall s \in L(P\| S)\ & \gamma(L(S), s) \bigcap \Sigma_c \subseteq \gamma(L(P), s) \\ \forall s \in L(P\| S)\ & out(\delta(S, s)) \bigcap \Sigma_c \subseteq out(\delta(P, s)) \end{aligned} \tag{3.14}$$

■

**Remark.** Again, note that the requirement of deterministic processes is essential here, just as in Lemma 3.3. However, note that it is *not* true that $L(S)$ being inverse controllable is equivalent to $L(P\| S)$ being inverse controllable. It holds, though, that if $L(S)$ is inverse controllable then $L(P\| S)$ is also inverse controllable. □

When does a supervisor have an inverse controllable language, and hence is inverse complete? For controllability (not inverse controllability) a sufficient condition would be that the language of $P$ is a subset of the language of $S$, that is $L(P) \subseteq L(S)$. In such a case it is obvious that $L(S)$ is controllable. For inverse controllability we have the following sufficient condition.

**Lemma 3.10** For a supervisor $S$ its language $L(S)$ is inverse controllable with respect to a plant $P$ and a set of controllable events $\Sigma_c \subseteq \Sigma_P$, if $L(S) \subseteq L(P)$.

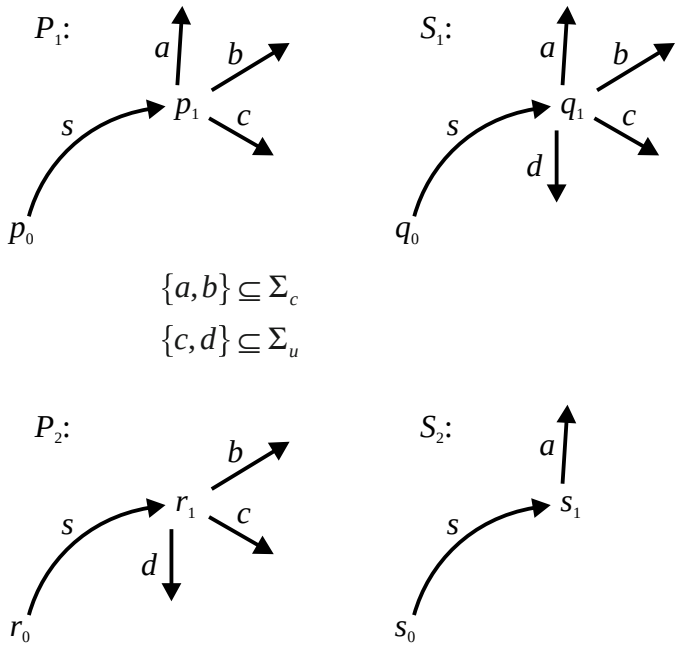



**Figure 3.2:** Illustration of completeness and inverse completeness. $S_1$ and $S_2$ are supervisors for the plants $P_1$ and $P_2$. $S_1$ is both complete and inverse complete with respect to $P_1$, since $out(p_1) \bigcap \Sigma_u \subseteq out(q_1)$ and $out(q_1) \bigcap \Sigma_c \subseteq out(p_1)$. Note though that $L(S_1)$ is *not* a subset of $L(P_1)$. $S_1$ is also complete with respect to $P_2$ but not inverse complete, and again $L(S_1) \not\subseteq L(P_2)$. $S_2$ is inverse complete with respect to $P_1$ but not with respect to $P_2$. $S_2$ is not complete with respect to any of the plants.

**Proof.** We have that $L(S) \subseteq L(P) \subseteq L(P)\Sigma_c$. Thus, $L(P)\Sigma_c \bigcap L(S) = L(S) \subseteq L(P)$. ■

**Remark.** Note that this is only a sufficient condition and is presented, without proof, by Balemi (1992), who also gives no necessary and sufficient conditions. The reason for $L(S) \subseteq L(P)$ not to be a necessary condition for supervisor inverse completeness, is that the definitions of inverse completeness and inverse controllability only concern the controllable events, the commands. As far as inverse completeness is regarded, the supervisor can choose to define any uncontrollable event whatsoever in its ready sets. Thus, the language of $S$ is not required to be included in the plant language. This is true even if we demand the supervisor to be complete, though then the supervisor must at least define the uncontrollable events defined by the plant. This is illustrated in Figure 3.2 below. □

Naturally, the notion of inverse completeness restricts the available solutions of the supervisor synthesis problem relative to the original SCT. It is also shown by Balemi (1992) that the solution set of the input/output interpretation is a subset of the solution set of the original Ramadge/Wonham approach. This is so, because now we have an extra restriction on the structure of the supervisor, and are not as free to choose the supervisor as without the constraint of inverse completeness. However, as Balemi (1992) also points out, since the solution to the supervisory synthesis problem as given by Theorem 3.6

generates a supervisor with a language that is in fact a subset of the plant language, this supervisor will always be inverse complete with respect to the plant. Note though, that in general we may choose controllable supervisor languages $L(S)$, such that $L(S) \not\subseteq L(P)$. Such languages are not necessarily inverse controllable.

Note that, when concerning completeness, the condition that $L(S) \subseteq L(P)$ arises naturally, since it is the behavior of the closed-loop system that is of concern, and in that behavior $P$ will always restrict the supervisor from going outside $L(P)$. On the other hand, with inverse completeness, where $S$ is considered to generate some events, this condition is explicit as a sufficient condition to ensure that the supervisor will not generate commands that the plant cannot accept.

When a supervisor is both complete and inverse complete with respect to a given plant, their synchronous composition is said to be *wellposed*. Balemi (1992) uses the prioritized synchronous composition to elegantly express the interconnections between plant and supervisor, as regards completeness, inverse completeness and wellposedness.

For a supervisor $S$ and a plant $P$, regarding completeness we have

$$P \| S = P_{\Sigma} \|_{\Sigma_u} S. \tag{3.15}$$

As regards inverse completeness we have

$$P \| S = P_{\Sigma_c} \|_{\Sigma} S. \tag{3.16}$$

And finally when the interconnection is wellposed

$$P \| S = P_{\Sigma_c} \|_{\Sigma_u} S. \tag{3.17}$$

The simplicity of these expressions is indeed elegant.

The input/output interpretation again raises the issue of event generation. A supervisor as defined by the Ramadge/Wonham approach would in the formalism of Milner (1989) only define complemented (overbarred) events, and a plant would only define non-complemented events. As defined by Balemi (1992), a supervisor would have the controllable events non-overbarred while the uncontrollable would be overbarred. For the plant, on the other hand, the situation would be the opposite.

Thus, let us denote by $\overline{\Sigma_S}$ and $\Sigma_S$ the complemented and non-complemented events of the supervisor, respectively, and likewise $\overline{\Sigma_P}$ and $\Sigma_P$ for the plant. Then we have for the Ramadge/Wonham approach $\overline{\Sigma_S} = \Sigma_c \bigcup \Sigma_u$, $\Sigma_S = \emptyset$, $\overline{\Sigma_P}$ and $\Sigma_P = \Sigma_c \bigcup \Sigma_u$, while for the input/output interpretation we have $\overline{\Sigma_S} = \Sigma_u$, $\Sigma_S = \Sigma_c$, $\overline{\Sigma_P} = \Sigma_c$ and $\Sigma_P = \Sigma_u$. Other variations are, of course, also conceivable.

### Summary

In this section we have briefly presented the input/output interpretation of the supervsiory control theory, as given by Balemi (1992). A supervisor is no longer just a passive device following the events generated by the plant and merely restricting event generation. The supervisor acts as a controller, receiving input from the plant and generating output to the plant. The events generated by the supervisor, the commands, are considered to be the controllable events, while the responses generated by the plant are the uncontrollable events.

In the next sections we will extend both the original supervisory control theory and the input/output interpretation to non-deterministic processes. This extension requires different approaches for the two interpretations of supervisor/plant interaction. For a supervisor as a passive device it is rather straight-forward, whereas for the input/output interpretation the added constraint of inverse completeness needs some more elaboration. In both cases, though, the properties of controllability and inverse controllability defined for a supervisor language will be shown to no longer be equivalent to the properties of completeness and inverse completeness of the supervisor automaton.

## 3.4 Non-deterministic Supervisory Control

The original Ramadge/Wonham formulation of the supervisory control theory considered only *deterministic* discrete event processes. These have inherent structural properties not applicable to the more general form of non-deterministic systems. Thus, the SCT implicitly included certain aspects given by the deterministic system that has to be explicitly spelled out for non-deterministic DES.

It is not until very recently, see Inan (1994), Overkamp (1994) and Shayman (1994), that the SCT has been generalized to non-deterministic systems. In Shayman (1994), though the plant is allowed to be non-deterministic, the specification of the behavior of the controlled system is given as a desired closed-loop *language*. Thus, the specification cannot be non-deterministic, since non-determinism is a property of a process, not of a language. Given that the specification language is controllable there is given necessary and sufficient conditions for there to exist a *deterministic* supervisor that can control the plant to meet the specification, marked as well as unmarked. Similarily, Inan (1994) uses a deterministic automaton as specification. In this case though, given a deterministic automaton representing a controllable language, a *non-deterministic* nonblocking supervisor is considered. Overkamp (1994) allows both the plant and the specification to be non-deterministic, and then a deterministic supervisor is generated, such that the closed-loop system *reduces* the specification. That is, the closed-loop system generates a sublanguage of the specification, and does not deadlock unless allowed by the specification. Thus, nonblocking is not considered by Overkamp (1994)[2]. We can note that in all of the works mentioned above, non-determinism arises as a consequence of some type of partial observation, representing model uncertainty. We can also note that none of the authors above have considered non-deterministic supervisors in conjunction with non-deterministic plants.

In this section we will examine non-deterministic supervisors. We will give necessary and sufficient conditions for the structure of a non-deterministic supervisor with a controllable language to be complete as well as inverse complete with respect to a non-deterministic plant. We will also show under what conditions a non-deterministic supervisor exists for a deterministic plant, given a non-deterministic specification, such that the closed-loop behaves as a subprocess of the specification.

Are non-deterministic supervisors of importance, one may ask. In the object oriented

[2]Note though, the unfortunate choice of Overkamp's terminology. Overkamp (1994) refers to deadlock as *blocking*, and consequently a deadlock-free system is called *nonblocking*. This is *not* nonblocking in the sense of Definition 2.22.

modeling of flexible manufacturing systems, as will be described, a non-deterministic specification arises naturally as individual product routes are composed into a global specification on the systems behavior. The product routes themselves are all deterministic, but the requirement for them to evolve mutually asynchronously leads to the non-deterministic specification. This is also the case in batch processes, as described in Tittus (1995b). In Tittus (1995d) is shown how such a non-deterministic specification results in a non-deterministic supervisor. Inan (1994) develops a theory to generate a non-deterministic supervisor from a deterministic one. The reason for this is that the deterministic supervisor relies on observing some unobservable events. The non-deterministic supervisor avoids this. These examples show that there are applications in which non-deterministic specifications and supervisors are necessary. Therefore, a rigid foundation for how to handle these problems is required.

### 3.4.1 Controllability, Completeness and Conformity

We begin by recapitulating and reformulating some of the expressions of controllability and completeness given in Section 3.2, and generalize some of these to non-deterministic processes. It is important to notice that the definitions and proofs given in Section 3.2 and those given in Ramadge (1987), Kumar (1991), Balemi (1992) and many others, depend on the fact that all automata are deterministic, even though this may not always be explicitly spelled out. When non-deterministic transition machines are considered, some of these definitions have to be reformulated, mainly those that concern the structure of the automaton. There does exist a certain structural property of non-deterministic automata that would make the original definitions and proofs hold, the notion of *conformity* described below.

Note that in this section we will make the two assumptions stated in Section 1.4, namely that, for a supervisor $S$ and a plant $P$ we have that $\Sigma_S = \Sigma_P$ and $L(S) \subseteq L(P)$. This is just for ease of notation and proofs, and does not incur any loss of generality. See the remark to Definition 3.1.

Those definitions involving only the language of an automaton are not altered when generalized to non-deterministic processes. Thus, the expression defining *controllability* is

$$L(S)\Sigma_u \bigcap L(P) \subseteq L(S) \tag{3.18}$$

which, by the definition of active events is equivalent to

$$\forall s \in L(P \,\|\, S) \;\; \gamma(L(P), s) \bigcap \Sigma_u \subseteq \gamma(L(S), s). \tag{3.19}$$

This expression is obviously equivalent to

$$\forall s \in L(P \,\|\, S) \;\; \gamma(L(P), s) \bigcap \Sigma_u \subseteq \gamma(L(S), s) \bigcap \Sigma_u. \tag{3.20}$$

This last expression, (3.20), will be used extensively in the following.

The definition of *nonblocking*, Definition 2.22 is also not altered since nonblocking is a language property.

However, *completeness* is a property of the transition machine that must be reformulated for non-deterministic automata. Naturally, such a reformulation must be a generalization of Definition 3.1, and contain this definition as a special case. Thus, we make the following definition.

**Definition 3.11 Completeness**

A supervisor $S$ is *complete* with respect to a plant $P$ and a set of uncontrollable events $\Sigma_u$ if and only if

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S,s)\ \forall p \in \delta(P,s)\ \mathrm{out}(p) \bigcap \Sigma_u \subseteq \mathrm{out}(q). \tag{3.21}$$

**Remark.** It is easy to verify that this definition is indeed a generalization of Definition 3.1 since when $S$ and $P$ are deterministic both $\delta(S,s)$ and $\delta(P,s)$ are singletons. In that case it is perfectly valid to write $\mathrm{out}(\delta(S,s))$, as in Definition 3.1, in place of $\forall q \in \delta(S,s)\ \mathrm{out}(q)$.

When $S$ is not complete, there exists a string $s \in L(P \,\|\, S)$, a state $q \in \delta(S,s)$ and a state $p \in \delta(P,s)$, such that $\mathrm{out}(p) \bigcap \Sigma_u \not\subseteq \mathrm{out}(q)$. In that case, $q$ is said to be *uncontrollable*. □

Completeness has been expressed in terms of the transition function, above. Naturally, it can also be expressed in terms of edges and traces, a version which will be used later. Then we have that a transition machine $S$ is complete with respect to a plant $P$ and a set of uncontrollable events $\Sigma_u$ if and only if for all $t_P \| t_S \in tr(P \,\|\, S)$ we have that

$$\left.\begin{array}{c} t_P e_P \in tr(P) \\ label(e_P) \in \Sigma_u \end{array}\right\} \Rightarrow \exists e_S \in E_S \text{ such that } label(e_S) = label(e_P) \text{ and } t_S e_S \in tr(S). \tag{3.22}$$

Here, $t_P \| t_S$ is used to denote the trace arisng in $tr(P \| S)$ due to synchronization of the traces $t_P \in tr(P)$ and $t_S \in tr(S)$. As was indicated by Section 3.2, completeness can be viewed as the fundamental requirement for a supervisor; unless the supervisor is complete with respect to the plant, we cannot guarantee that the behavior of the closed-loop system is contained within the specification. It was shown by Lemma 3.3 that completeness and controllability are equivalent *when considering deterministic processes*. However, when non-determinism is involved this does not hold entirely.

It will be shown below that a complete supervisor $S$ will always represent a controllable language $L(S)$, but the converse is not necessarily true. This so, since the notion of controllability is weaker than the notion of completeness in the case of non-deterministic supervisor. Controllability concerns the active events, while completeness concerns the ready sets of the supervisor. For a given supervisor state reached by a string $s$, the ready set is a subset of the active event set corresponding to $s$, as shown by the remark to Definition 2.25. See also Figure 2.2 on Page 33.

**Lemma 3.12** A supervisor $S$ that is complete with respect to a plant $P$, with a set of uncontrollable events $\Sigma_u$, will always have a controllable language $L(S)$.

**Proof.** Completeness is defined as

$$\forall s \in L(P \,\|\, S)\ \ \forall p \in \delta(P,s)\ \ \forall q \in \delta(S,s)\ \ \mathrm{out}(p) \bigcap \Sigma_u \subseteq \mathrm{out}(q), \tag{3.23}$$

which, by Lemma A.1 can be rewritten as

$$\forall s \in L(P \,\|\, S)\ \ \forall p \in \delta(P,s)\ \ \forall q \in \delta(S,s)\ \ \mathrm{out}(p) \bigcap \Sigma_u \subseteq \mathrm{out}(q) \bigcap \Sigma_u. \tag{3.24}$$

Now, we can use Lemma A.4 with $f(\cdot) \equiv \text{out}(\cdot) \bigcap \Sigma_u$ to write (3.24) equivalently as

$$\forall s \in L(P \,\|\, S) \;\; \forall q \in \delta(S, s) \;\; \gamma(L(P), s) \bigcap \Sigma_u \subseteq \text{out}(q) \bigcap \Sigma_u. \tag{3.25}$$

Finally, by Lemma A.5 (3.25) *implies* that

$$\forall s \in L(P \,\|\, S) \;\; \gamma(L(P), s) \bigcap \Sigma_u \subseteq \gamma(L(S), s) \bigcap \Sigma_u, \tag{3.26}$$

which by (3.20) means that $L(S)$ is controllable. ∎

**Remark.** Note that this also means that when $S$ is complete, then $L(P \,\|\, S)$ is controllable. This holds whether the plant and supervisor are deterministic or not, since the properties of the languge represented by an automaton is not altered due to nondeterminism. Thus, controllability of $L(S)$ is equivalent to controllability of $L(P \,\|\, S)$. □

Note again that (3.25) and (3.26) are *not* equivalent, when $S$ is allowed to be nondeterministic. This is a consequence of the fact that for a non-deterministic $S$, the ready set of a state reached by a string $s$, is not equal to the active set after $s$. Thus, controllability of $L(S)$ is not sufficient for $S$ to be complete; a fact which we emphasize by the following corollary.

**Corollary 3.13** A supervisor $S$ with a controllable language $L(S)$ is *not necessarily* complete, with respect to a plant $P$ and an alphabet of uncontrollable events $\Sigma_u$.

**Remark.** Note though, that a supervisor cannot be complete if its language is not controllable. This is because completeness is a stronger property than controllability. The set of all complete supervisors is a subset of the set of all supervisors with controllable language. Thus, if we choose a supervisor outside the set of the ones with controllable languages, we will never choose a complete supervisor. □

That for a deterministic supervisor controllability is equivalent to completeness is due to the fact that for all strings the active event set is equal to the ready set of the state reached by that string. Thus, the equivalence of controllability of $L(S)$ and completeness of $S$, does not depend on the determinism of $P$. This fact is emphasised in the following corollary to Lemma 3.12 which is a reformulation of Lemma 3.3.

**Corollary 3.14** A *deterministic* supervisor $S$ is complete with respect to a, possibly non-deterministic, plant $P$, with a set of uncontrollable events $\Sigma_u$, if and only if $L(S)$ is controllable.

Note that Corollary 3.14 only concerns deterministic *supervisor*. The plant $P$ can be non-deterministic, since when $S$ is deterministic and $L(S)$ is controllable, the state reached after string $s$ will have a ready set that includes all the uncontrollable events that $P$ defines in any state reached by $s$. See also Example 3.1.

For controllability to be equivalent to completeness we have to add some restriction on the structure of $S$. For $S$ to be deterministic is a sufficient but not necessary condition. In fact, it is enough to require $S$ to have equal ready sets of the states of $\delta(S, s)$, as far

**Example 3.1 Controllable, Complete and Conforming Supervisors**

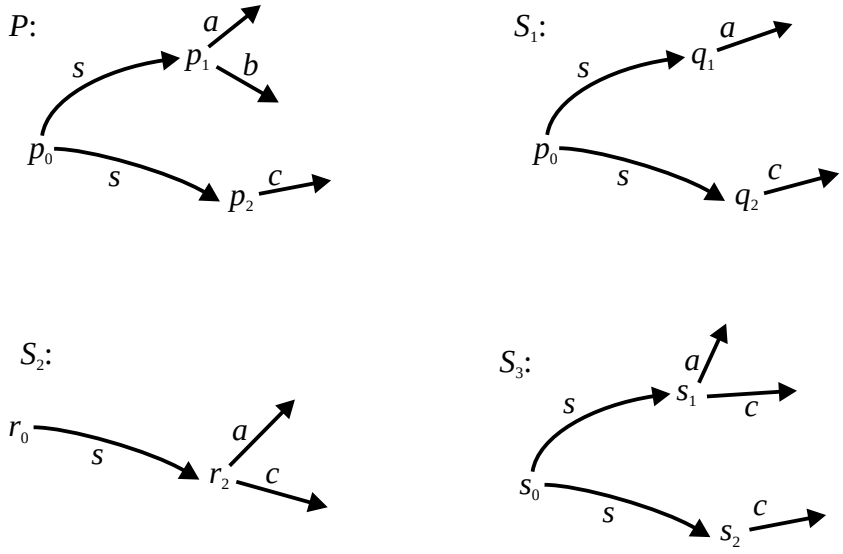



**Figure 3.3:** Examples of controllability, completeness and conformity of non-deterministic supervisors, $S_1$, $S_2$ and $S_3$, with respect to a non-deterministic plant $P$.

In Figure 3.3 is shown three supervisors, $S_1$, $S_2$ and $S_3$, for a plant $P$. The languages of all three supervisors are equal, $L(S_1) = L(S_2) = L(S_3) = \{\varepsilon, s, sa, sc\}$ and the plant language is $L(P) = \{\varepsilon, s, sa, sb, sc\}$. The uncontrollable event is $\Sigma_u = \{c\}$. The supervisor languages are controllable since $L(S_i)\Sigma_u \bigcap L(P) = \{sc\} \subseteq L(S_i)$ (for $i = 1, 2, 3$). However, $S_1$ is not complete with respect to $P$, since $\delta(P, s) = \{p_1, p_2\}$ and $\delta(S_1, s) = \{q_1, q_2\}$ and $\text{out}(p_2) \bigcap \Sigma_u = \{c\} \not\subseteq \text{out}(q_1) = \{a\}$. Both $S_2$ and $S_3$ are complete, on the other hand, since, both $\text{out}(p_1) \bigcap \Sigma_u = \emptyset$ and $\text{out}(p_2) \bigcap \Sigma_u$ are subsets of $\text{out}(r_1) = \{a, c\}$, $\text{out}(s_1) = \{a, c\}$ and $\text{out}(s_2) = \{c\}$. Note also that $S_1$ is not conforming, while both $S_2$ and $S_3$ are.

■

as $\Sigma_u$ is regarded. We can think of this as $S$ must be deterministic with regard to $\Sigma_u$. This restriction will be called *conformity* and a supervisor with this property is said to be *conforming* with respect to $\Sigma_u$. The formal definition looks like this.

**Definition 3.15 Conforming Supervisor**

A supervisor $S$ is said to be *conforming* with respect to a plant $P$ with uncontrollable events $\Sigma_u$, if and only if all states reachable by one and the same string defines equal ready sets with regard to $\Sigma_u$. That is,

$$S \text{ conforming} \Leftrightarrow \forall s \in L(P \,\|\, S)\ \forall q_1, q_2 \in \delta(S, s)\ \text{out}(q_1) \bigcap \Sigma_u = \text{out}(q_2) \bigcap \Sigma_u. \tag{3.27}$$

**Remark.** When $S$ is deterministic, $\delta(S, s)$ is a singleton and so $S$ is always conforming. □

By using Lemma A.12 we immediately get the following equivalent expression for conformity,

$$\forall s \in L(P \,\|\, S)\ \ \forall q \in \delta(S, s)\ \ \gamma(L(S), s) \bigcap \Sigma_u = \text{out}(q) \bigcap \Sigma_u. \tag{3.28}$$

By combining (3.28) with (3.26) it follows that when $L(S)$ is controllable and $S$ is conforming with respect to $\Sigma_u$, then $S$ is also complete. Additionally, we know that when $S$ is complete, $L(S)$ is always controllable. This is summarized in the following theorem.

**Theorem 3.16** For a plant $P$ with a set of uncontrollable events $\Sigma_u$, a supervisor $S$ that is conforming with respect to $\Sigma_u$, is complete if and only if $L(S)$ is controllable. That is,

$$S \text{ conforming} \Rightarrow (S \text{ complete} \leftrightarrow L(S) \text{ controllable}). \tag{3.29}$$

**Proof.** It is obviously so that (3.25) and (3.26) are equivalent when (3.28) holds. ■

**Remark.** Note that $S$ being deterministic is a special case of $S$ being conforming. In Example 3.2 is illustrated that a non-deterministic supervisor with a controllable language is not necessarily complete, unless it is conforming, that is. □

We illustrate the relations between completeness and controllability in Figure 3.4. From completeness we can always get to controllability and conformity. However, to get from controllability to completeness, we must cross the gap by means of the gray arrow. This is where the additional requirement of $S$ being conforming comes in. When $S$ is conforming, such as when it is deterministic, then this gap is bridged, and completeness and controllability are equivalent.

Finally, it can be interesting to note that when a supervisor is complete and its language is a subset of the plant language, then that supervisor is always conforming.

**Lemma 3.17** For a plant $P$, with a set of uncontrollable events $\Sigma_u$, and a supervisor $S$ with $L(S) \subseteq L(P)$, $S$ complete implies $S$ conforming with respect to $\Sigma_u$.

**Proof.** Since $L(S) \subseteq L(P)$, we can replace $L(P \,\|\, S)$ by $L(S)$.

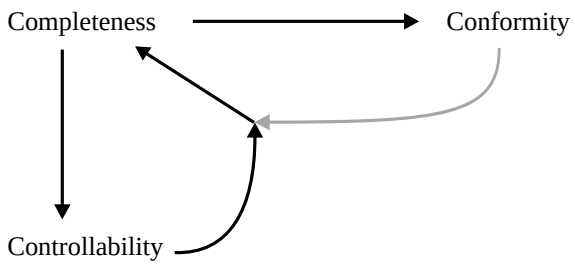



**Figure 3.4:** Relations between completeness, controllability and conformity. The gray arrow means that conformity is required to cross the gap from controllability completeness. Compare also Figure A.1.

By Corollary 2.27

$$L(S) \subseteq L(P) \Rightarrow \forall s \in L(S)\ \gamma(L(S), s) \bigcap \Sigma_u \subseteq \gamma(L(P), s) \bigcap \Sigma_u. \tag{3.30}$$

By (3.25) we can write completeness as

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S, s)\ \gamma(L(P), s) \bigcap \Sigma_u \subseteq \mathrm{out}(q) \bigcap \Sigma_u. \tag{3.31}$$

Combining these two expressions, we get

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S, s)\ \gamma(L(S), s) \bigcap \Sigma_u \subseteq \gamma(L(P), s) \bigcap \Sigma_u \subseteq \mathrm{out}(q) \bigcap \Sigma_u, \tag{3.32}$$

meaning that the union of all $\mathrm{out}(q) \bigcap \Sigma_u$ is a subset of every $\mathrm{out}(q) \bigcap \Sigma_u$. Of course, this can only happen if all $\mathrm{out}(q) \bigcap \Sigma_u$ are equal, so that

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S, s)\ \gamma(L(S), s) \bigcap \Sigma_u = \mathrm{out}(q) \bigcap \Sigma_u, \tag{3.33}$$

thus proving Lemma 3.17. ∎

As mentioned in Section 3.2, there is really no loss of generality to assume that $L(S) \subseteq L(P)$. Using Lemmas 3.12 and 3.17 we get the following theorem, realting completeness, controllability and conformity.

**Theorem 3.18** A supervisor $S$, for a plant $P$ with a set of uncontrollable events $\Sigma_u$, with $L(S) \subseteq L(P)$, is complete if and only if $L(S)$ is controllable and $S$ is conforming. That is,

$$L(S) \subseteq L(P) \Rightarrow (S \text{ complete} \leftrightarrow L(S) \text{ controllable} \wedge S \text{ conforming}), \tag{3.34}$$

all with respect to $\Sigma_u$

It is interesting to note the following words, (here $K = L(P \| S) = L(P) \bigcap N$ and $det(K)$ is a deterministic automaton with language $K$).

> *... when the supervisor is chosen to be $det(K)$. The determinism of $S$ is essential here. If $S$ is a non-deterministic* [supervisor] *with $L(S) = N$, there is* no guarantee *that the closed-loop language* $[\, L(P \| S)]$ *will be $K$.*
>
> Shayman (1994), Remark 11

**Example 3.2 Conforming and Nonconforming Supervisors**

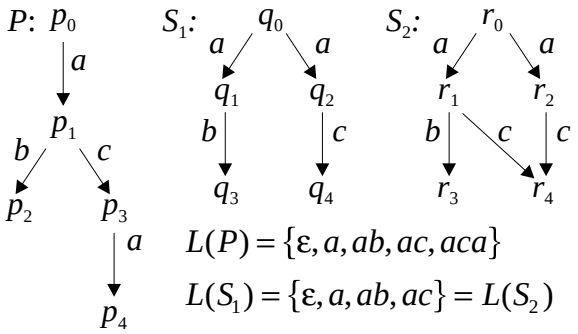



**Figure 3.5:** A deterministic plant, $P$, and two supervisors, $S_1$ and $S_2$. The languages of the supervisors are equal and controllable, but $S_1$ is not conforming and therefore not complete with respect to $\Sigma_u = \{c\}$.

This example shows that of two supervisors with equal and controllable languages, only the one that is conforming is also complete with respect to the plant. The only uncontrollable event is $\Sigma_u = \{c\}$. As can be seen in Figure 3.5, the two supervisors, $S_1$ and $S_2$ have the same language. It is easily verified that this language is indeed controllable with respect to $L(P)$ and $\Sigma_u$. However, $S_1$ is not complete since it cannot follow the uncontrollable event c when in its $q_1$ state. $S_2$, on the other hand, can follow the uncontrollable event in both its $r_1$ and $r_2$ states. Thus, $S_2$ has both a controllable language and is conforming with respect to $\Sigma_u$. Therefore $S_2$ is also complete, which is evident from Figure 3.5.

■

This observation has above been shown to be accurate, since a non-deterministic supervisor guarantees the controllable specification language *only when it is conforming with respect to the uncontrollable events*. Note also that this holds regardless of whether the plant is non-deterministic or not. Example 3.2, adapted from Shayman (1994), shows that a non-deterministic supervisor is not necessarily complete when its language is controllable even though the plant is deterministic. Not unless the supervisor is conforming, that is.

Finally, we will generalize Lemma 3.4 to non-deterministic processes. This is needed in the proof of existence of non-deterministic supervisors, given non-deterministic specifications, in Section 3.5. However, when a plant $P$ is non-deterministic, it is not necessarily complete with respect to itself. This is obvious from the general definition of completeness, Definition 3.11. Not unless all the states reached by a string $s$ defines the same set of uncontrollable events, that is. This condition is satisfied when the plant is deterministic, for example. But more generally, it is satisfied when the plant is conforming with respect to the uncontrollable events. Compare Definition 3.15 and see also Definition 3.21.

**Theorem 3.19** For a conforming plant $P$ and a supervisor $S$, the closed-loop system $P\|S$ is complete with respect to $P$ if and only if $S$ is complete with respect to $P$. All with respect to a set of uncontrollable events $\Sigma_u \subseteq \Sigma_P$. That is,

$$P \text{ conforming} \Rightarrow (\ P\|S \text{ complete} \Leftrightarrow S \text{ complete}). \tag{3.35}$$

**Proof.** When the above conditions hold, the following four expressions are all equivalent

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S,s)\ \forall p \in \delta(P,s)\ \mathrm{out}(p) \bigcap \Sigma_u \subseteq \mathrm{out}(q). \tag{3.36}$$

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S,s)\ \forall p,p' \in \delta(P,s)\ \mathrm{out}(p') \bigcap \Sigma_u \subseteq \mathrm{out}(p) \bigcap \mathrm{out}(q) \bigcap \Sigma_u. \tag{3.37}$$

$$\forall s \in L(P \,\|\, S)\ \forall \langle p,q\rangle \in \delta(P\|S,s)\ \forall p' \in \delta(P,s)\ \mathrm{out}(p') \bigcap \Sigma_u \subseteq \mathrm{out}(p) \bigcap \mathrm{out}(q) \bigcap \Sigma_u \tag{3.38}$$

$$\forall s \in L(P\|(P\|S))\ \forall \langle p,q\rangle \in \delta(P\|S,s)\ \forall p' \in \delta(P,s)\ \mathrm{out}(p') \bigcap \Sigma_u \subseteq \mathrm{out}(\langle p,q\rangle). \tag{3.39}$$

■

**Remark.** Equation (3.36) expresses completeness of $S$ by Definition 3.11. That (3.36) is equivalent to (3.37) comes from the fact that $P$ is conforming with respect to $\Sigma_u$. Then specifically for the strings of $L(P \,\|\, S)$ it holds that the ready sets of the states reached by $s$ are all equivalent with respect to $\Sigma_u$. Since $S$ is complete, no uncontrollable events are removed from $P \,\|\, S$. Therefore, (3.38) is equivalent to (3.37). Finally, by using the fact that, since $\Sigma_S = \Sigma_P$, for all states $\langle p,q\rangle \in Q_{P\|S}$ it holds that $\mathrm{out}(\langle p,q\rangle) = \mathrm{out}(p) \bigcap \mathrm{out}(q)$, see (2.53), and that the languages $L(P\|S)$ and $L(P\|(P\|S))$ are equivalent by Corollary 2.45 we get the final equivalence. Obviously, (3.39) shows that $P\|S$ is in fact complete with respect to $P$ and $\Sigma_u$.

Naturally, similar reasoning shows the converse. The equivalence between (3.38) and (3.37) comes from $P \,\|\, S$ being complete, so that no uncontrollable events have been lost in the synchronization. □

**Example 3.3 Inverse Controllability but not Inverse Completeness**

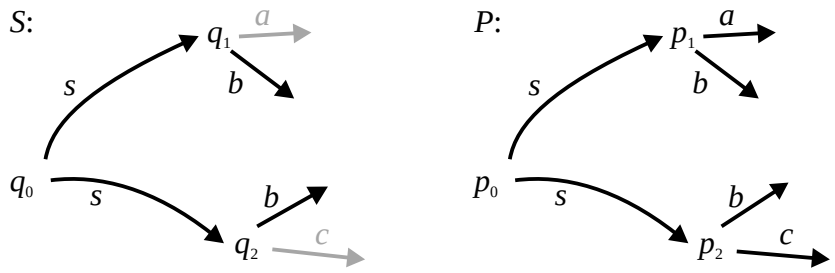



**Figure 3.6:** An example of a supervisor and plant, where the supervisor is not inverse complete, even though its language is inverse controllable according to Definition 3.8.

In Figure 3.6 is shown a supervisor $S$ with a language $L(S)$ that is inverse controllable, since $L(S) \subseteq L(P)$. All shown events are considered to be controllable, that is, $\{a, b, c\} \subseteq \Sigma_c$. After the string $s$, the plant $P$ can be in either $p_1$ or $p_2$, while $S$ can be in either $q_1$ or $q_2$. If, which is considered to be the case, the supervisor cannot determine which state $P$ actually occupies, for $S$ to be inverse complete it must never generate either the $a$ or $c$ commands when in either of the states $q_1$ or $q_2$. If the supervisor did execute command $a$ when the plant is in state $p_2$, the notion of inverse completeness would be violated. Thus, $S$ is not inverse complete with respect to the given plant even though $L(S)$ is inverse controllable. This so, since (from (3.44)) $\gamma(L(S), s) \bigcap \Sigma_c = \{a, b, c\} \not\subseteq \bigcap_{p \in \delta(P,s)} \text{out}(p) = \{b\}$.

■

### 3.4.2 Inverse Properties

When regarding non-deterministic processes, it is not surprising that the notion of inverse completeness has to be rephrased. It is also not surprising that inverse controllability of $L(S)$ is no longer equivalent to $S$ being inverse complete. The duality of controllability/inverse controllability and completeness/inverse completeness carries these facts with it.

In the general setting of non-deterministic plant and supervisor, there may exist several states reachable by the same string $s$, each of which may have different ready sets. Naturally, if the supervisor $S$ is not to generate any commands that the plant cannot follow, the set of controllable events of the ready set of *every* state reachable by $s$ in $S$ must be a subset of the ready sets of *all* states reached by $s$ in $P$. See Example 3.3. Thus we can make the following definition of inverse completeness.

**Definition 3.20 Inverse Completeness**

A supervisor $S$ is said to be *inverse complete* with respect to a plant $P$ and a set of controllable events $\Sigma_c$, if and only if it only generates controllable events (commands) that $P$ can accept. That is,

$$S \text{ inverse complete} \Leftrightarrow \forall s \in L(P \,\|\, S) \;\; \forall p \in \delta(P, s) \;\; \forall q \in \delta(S, s) \;\; \text{out}(q) \bigcap \Sigma_c \subseteq \text{out}(p). \tag{3.40}$$

**Remark.** It is obvious that this definition is more general than the definition given for deterministic supervisors, Definition 3.7. The difference being that we no longer assume that one and the same string reaches a single state. Note though that the informal wording is the same, only the formal expression has been reformulated. □

Let us now use Lemma A.1 and Lemma A.7 to write equivalent expressions of inverse completeness. Definition 3.20 is equivalent to the following expressions.

By Lemma A.1, we have

$$\forall s \in L(P \,\|\, S)\ \ \forall p \in \delta(P,s)\ \ \forall q \in \delta(S,s)\ \ \mathrm{out}(q) \bigcap \Sigma_c \subseteq \mathrm{out}(p) \bigcap \Sigma_c, \tag{3.41}$$

which is by Lemma A.7, with $f(\cdot) \equiv \mathrm{out}(\cdot) \bigcap \Sigma_c$ equivalent to

$$\forall s \in L(P \,\|\, S)\ \forall p \in \delta(P,s)\ \gamma(L(S),s) \bigcap \Sigma_c \subseteq \mathrm{out}(p) \bigcap \Sigma_c. \tag{3.42}$$

But by Lemma A.7, (3.41) is also equivalent to

$$\forall s \in L(P \,\|\, S)\ \forall q \in \delta(S,s)\ \ \mathrm{out}(q) \bigcap \Sigma_c \subseteq \bigcap_{p \in \delta(P,s)} \mathrm{out}(p) \bigcap \Sigma_c. \tag{3.43}$$

**Remark.** We can note that (3.42) is the immediate dual of (3.25). Just interchange $p$ and $q$, and replace $\Sigma_u$ with $\Sigma_c$.

The expression (3.43) can also be seen as a dual of (3.25). In this case, also the union $\gamma(L(S),s) = \bigcup_{q \in \delta(S,s)} \mathrm{out}(q)$ is replaced by the intersection $\bigcap_{p \in \delta(P,s)} \mathrm{out}(p) \bigcap \Sigma_c$. More importantly though, (3.43) is presented, since we believe it to be a more intuitive description of inverse completeness than (3.42). In the case of completeness, the supervisor must be guaranteed to follow all uncontrollable events generated by the plant; that is, the union of all feasible uncontrollable plant events. On the other hand, with inverse completeness, the supervisor must be guaranteed to only generate commands that the plant can follow; that is, the intersection of all feasible controllable plant events. See also Example 3.3.

We can also note that inverse completeness is equivalent to

$$\forall s \in L(P \,\|\, S)\ \gamma(L(S),s) \bigcap \Sigma_c \subseteq \bigcap_{p \in \delta(P,s)} \mathrm{out}(p) \bigcap \Sigma_c, \tag{3.44}$$

by Lemma A.11. An expression that might suggest an efficient way to check for inverse completeness. See Example 3.3. □

We can also note that by Lemma A.9 we have that inverse completeness *implies* that

$$\forall s \in L(P \,\|\, S)\ \gamma(L(S),s) \bigcap \Sigma_c \subseteq \gamma(L(P),s) \bigcap \Sigma_c, \tag{3.45}$$

which is in fact inverse controllability. Compare (3.26). As a dual to Lemma 3.12, (3.45) above shows that inverse completeness implies inverse controllability.

In Theorem 3.16 we used the notion of conformity to bridge the gap between controllability and completeness for non-deterministic systems. We also showed that completeness of $S$ implies conformity of $S$, but recall that this proof involved the assumption that $L(S) \subseteq L(P)$. When regarding inverse completeness and conformity, this assumption cannot help us in showing that inverse completeness implies conformity. In fact, we cannot show this at all.

Moreover, it is not conformity of $S$ that is of significance here, but a similar restriction on the plant, $P$.

**Definition 3.21 Conforming Plant**

A plant $P$ is said to be *conforming* with respect to a supervisor $S$, with a set of controllable (command) events $\Sigma_c$, if and only if all states reachable by one and the same string of the closed-loop system defines equal ready sets with regard to $\Sigma_c$. That is,

$$P \text{ conforming} \Leftrightarrow \forall s \in L(P \parallel S)\ \forall p_1, p_2 \in \delta(P, s)\ \text{out}(p_1) \bigcap \Sigma_c = \text{out}(p_2) \bigcap \Sigma_c. \quad (3.46)$$

**Remark.** When $P$ is deterministic, $\delta(P, s)$ is a singleton and so $P$ is always conforming.

Note that a plant $P$, as well as a supervisor $S$, can be conforming with respect to any subset of events of $\Sigma_P = \Sigma_S$. When the context unambigously tells us which set of events we mean, we will simply say that $P$ or $S$ is conforming, as in (3.46). The event-sets that will be of interest are, of course, $\Sigma_u$ and $\Sigma_c$. □

However, it is not realistic to demand $P$ to be conforming, since we do not choose $P$. We can only guarantee that $S$ is conforming with respect to the plant, since we create $S$. However, *if* $P$ is conforming *then* we can show that inverse controllability is equivalent to the definition of inverse completeness.

**Lemma 3.22** For a plant $P$ and a supervisor $S$, inverse controllability is equivalent to inverse completeness, if the plant is conforming. That is,

$$P \text{ conforming} \Rightarrow (S \text{ inverse complete} \leftrightarrow L(S) \text{ inverse controllable}). \quad (3.47)$$

**Proof.** Conformity of the plant can, by Lemma A.12, be equivalently rewritten as

$$\forall s \in L(P \parallel S)\ \forall p \in \delta(P, s)\ \text{out}(p) \bigcap \Sigma_c = \bigcap_{p \in \delta(P,s)} \text{out}(p) \bigcap \Sigma_c; \quad (3.48)$$

an expression that, by inspection of the expressions (3.44) and (3.45), immediately shows that when the plant is conforming, inverse controllability and inverse completeness are equivalent. ■

**Remark.** In the case of completeness and controllability, conformity of the supervisor was a necessary and sufficient condition for completeness to be equivalent to controllability. This so, since we were able to derive that a complete supervisor $S$ is always conforming when $L(S) \subseteq L(P)$. In the case of inverse controllability and inverse completeness, things are not so easy. A conforming plant is just a sufficient condition, since the inverse controllability and inverse completeness does not induce any conditions on $P$; merely on $S$ itself.

In the deterministic case, $L(S) \subseteq L(P)$ was shown to be a sufficient condition for $S$ to be inverse complete, again, not a necessary condition. The problem is that in the case of completeness, $P$ generated the events and the demand was on $S$. Naturally, this tightly links $S$ to $P$. In the case of inverse completeness, $S$ generates (some) events, but the demands still lie on $S$. Thus, $P$ is not linked to $S$. □

To bring structure into the relations between inverse completeness and inverse controllability, we show Figure 3.7. With deterministic $P$ and $S$, all expressions are equivalent,

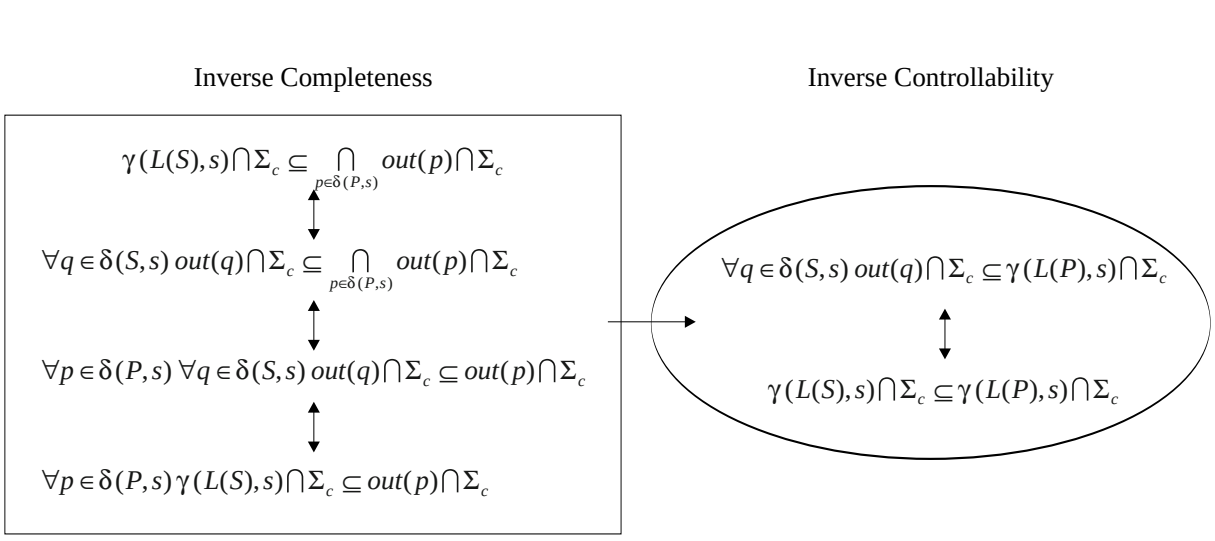



**Figure 3.7:** A systematic representation of the correspondance between the various equations given for inverse completeness and inverse controllability. Compare also Figure A.1, as well as Figure 3.4.

since then both $\delta(P,s)$ and $\delta(S,s)$ are singletons. Herein lies the root to the difference of complexity between deterministic and non-deterministic systems. However, we can be more general than this. Determinism is a specialization of conformity. A sufficient condition for inverse controllability to be equivalent to inverse controllability is that the plant is conforming, for example, deterministic.

For non-deterministic systems, inverse controllability at the right is not equivalent to inverse completeness at the left, because of the implication between the two sets of expressions. Thus, for non-deterministic systems we have to have some added constraint to bridge the gap. It has been shown that one possible added constraint is the property of plant conformity.

Note that with the above definition of inverse completeness (as well as completeness) a plant is seldom inverse complete (or complete) with respect to itself. Only when all its ready sets for states reachable by one and the same string are equal with respect to $\Sigma_c$ ( $\Sigma_u$), will this be the case. This is, of course, conformity.

### 3.4.3 Blocking and Nonblocking

In the previous sections we have shown that with a non-deterministic supervisor, completeness and controllability are no longer equivalent. Not unless the supervisor is conforming, that is. Similarily, inverse completeness and inverse controllability are also not necessarily equivalent. In the case where the plant is conforming, though, they are. Another aspect of interest for supervisory control, is the nonblockingness of the closed-loop system. We do not want to allow the system to halt in a nonmarked state, or even to transit indefinitely between two unmarked states, even though the supervisor may be both complete and inverse complete. We want the closed-loop system to be nonblocking.

However, nonblocking is a language property, saying that $\overline{L_m(P \,\|\, S)} = L(P \,\|\, S)$. That is, every string of $L(P \,\|\, S)$ can be continued to reach a marked state. By the

**Example 3.4 Nonblocking Language but Blocking Transition Machine**

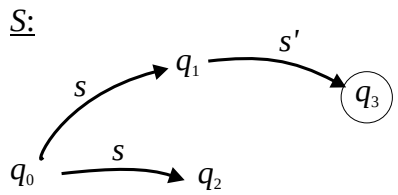



**Figure 3.8:** A transition machine with a nonblocking language, but a non-coaccessible state.

In Figure 3.8 is shown a transition machine, $S$, with a nonblocking language. The marked language is $L_m(S) = \{ss'\}$, so that $\overline{L_m(S)} = \{\varepsilon, s, ss'\}$. The closed language is, of course, $L(S) = \{\varepsilon, s, ss'\}$, so that $\overline{L_m(S)} = L(S)$ and $S$ is nonblocking. However, note that from the state $q_2$, no marked state can be reached, thus, $q_2$ is not coaccessible.

■

definition of the marked language of a transition machine, Definition 2.22, a string is included in the marked language if it reaches *some* marked state. This is, in fact, not strong enough to guarantee that, when the plant and the supervisor are both allowed to be non-deterministic, the system can always reach a marked state. Though the language is nonblocking, there may exist non-coaccessible states reached by strings that reach a marked state by some other route through the transition machine, see Example 3.4.

Thus, when non-determinism is involved, the intuitive notion of nonblockingness, regarding languages, is not adequate. In the light of the two previous sections, this is not a surprising fact. The problem is that with non-deterministic systems, one and the same string may lead to several states. The condition that $\overline{L_m(\cdot)} = L(\cdot)$ only requires that a marked state be reached from *at least one* of these states, as opposed to *all* of these states. We make the following definition.

**Definition 3.23 Trace Nonblocking**

A transition machine $S$ is *trace nonblocking*, if and only if the prefix closure of its marked traceset is equal to its closed traceset. That is,

$$S \text{ trace nonblocking} \Leftrightarrow \overline{tr_m(S)} = tr(S). \tag{3.49}$$

**Remark.** In the remark to Definition 2.22 we showed that the following implications hold for a transition machine $S$,

$$S \text{ trim} \Rightarrow \overline{tr_m(S)} = tr(S) \Rightarrow \overline{L_m(S)} = L(S). \tag{3.50}$$

Thus, when $S$ is trim, it is also trace nonblocking, and whenever it is trace-nonblocking it is also language nonblocking.

See also Kumar (1994), who define *trajectory model non-blocking*, which requires that each trajectory belonging to the closed trajectory set be extendable to a trajectory of the

marked trajectory set. This lies in-between trace nonblocking and language nonblocking. □

When a transition machine $S$ is trace nonblocking, all accessible states are also coaccessible. That is

$$\forall s \in L(S)\, \forall q \in \delta(S,s)\, \exists s' \in \Sigma_S^* \text{ such that } \delta_S(q,s') \bigcap M_S \neq \emptyset. \tag{3.51}$$

Note that, for *all* states $q$, reached by a string $s$, there is required to exist *some* string $s'$, reaching from $q$ to some marked state. It is not required that all strings emanating from $q$ reach marked states. Also, it is only required that *some* state reached from $q$ by $s'$ is marked. This must hold for all states reached by $s$.

For language-nonblocking, the corresponding expression looks like

$$\forall s \in L(S)\, \exists q \in \delta(S,s)\, \exists s' \in \Sigma_S^* \text{ such that } \delta_S(q,s') \bigcap M_S \neq \emptyset. \tag{3.52}$$

Here it is only required that for *some* state $q$, reached by a string $s$, there exists *some* string $s'$, reaching some marked state from $q$. All states reachable from $q$ are not required to be marked.

Comparing the two expressions, it is easy to see that that (3.51) implies (3.52), and that they are equivalent, if and only if $S$ is deterministic. Because of this equivalence, when we speak of determinsitic transition machines, we will merely call them *blocking* or *nonblocking*, whatever the case may be.

Assume now that we have a nonmarked plant $P$ and a marked complete supervisor $S$, with $\Sigma_P = \Sigma_S$. The closed-loop system is then by definition trace-nonblocking, if and only if

$$\forall s \in L(P \,\|\, S)\, \forall \langle p,q \rangle \in \delta(P \,\|\, S, s)\, \exists s' \in \Sigma_{P\,\|S}^* \text{ such that } \delta_{P\,\|S}(\langle p,q \rangle, s') \bigcap M_{P\,\|S} \neq \emptyset. \tag{3.53}$$

We have that $\delta_{P\|S}(\langle p,q \rangle, s) = \delta_P(p,s) \times \delta_S(q,s)$, see (2.49) of Definition 2.43. Thus, (3.53) can be equivalently expressed as

$$\forall s \in L(P \,\|\, S)\, \forall \langle p,q \rangle \in \delta(P \,\|\, S, s)\, \exists s' \in \Sigma_P^*\ [\delta_P(p,s') \times \delta_S(q,s')] \bigcap M_{P\,\|S} \neq \emptyset. \tag{3.54}$$

Since $P$ is nonmarked, $M_P = Q_P$, so that $M_{P\,\|S} = Q_P \times M_S$ by definition. Furthermore, it holds that $[A \times B] \bigcap [C \times D] = [A \bigcap C] \times [B \bigcap D]$, and, by definition $A \times \emptyset = \emptyset$ for any set $A$. Thus we can write

$$\forall s \in L(P \,\|\, S)\, \forall \langle p,q \rangle \in \delta(P \,\|\, S, s)\, \exists s' \in \Sigma_P^*\ [\delta_P(p,s') \bigcap Q_P] \times [\delta_S(q,s') \bigcap M_S] \neq \emptyset. \tag{3.55}$$

If there exists a string $s'$ such that $ss' \in L(P \,\|\, S)$, then it is obvious that $P \,\|\, S$ is trace nonblocking whenever $S$ is. Since then $\delta_P(p,s')$ and $\delta_S(q,s')$ are both defined, and $\delta_S(q,s') \bigcap M_S \neq \emptyset$. In fact, $S$ is not required to be trace nonblocking in its entirety, merely with regard to the language of the closed-loop system. However, $S$ trace nonblocking is a sufficient condition for $P \,\|\, S$ to be trace nonblocking, if that string $s'$ exists.

However, it may happen that there does not exist any string $s'$ at all, such that $\delta_P(p,s')$ and $\delta_S(q,s')$ are both defined. In such a case, the closed-loop system would have a terminating state $\langle p,q \rangle$. Such a terminating state results from the plant branching one

way and the supervisor another while executing a string $s$, until there are no more mutual events to execute. This is a consequence of allowing both the plant and the supervisor to be non-deterministic. In that case, unless that state $\langle p, q \rangle$ is marked, the closed-loop system cannot be trace nonblocking.

What are the requirements for such states to always be marked in the closed-loop system?

Naturally, all terminating states being marked is a necessary requirement for a transition machine to be trace nonblocking. It is not sufficient, though, since, for instance, an accessible nonmarked state that can only reach itself is not terminating, but it is also not coaccessible. Such a *livelock* is precisely the reason that the approach of Overkamp (1994) does not generate a nonblocking closed-loop system.

When composing a non-deterministic supervisor and a non-deterministic plant under full sycnhronous composition, we are interested in under what conditions the closed-loop system is trace nonblocking. The problem is that when both the supervisor and the plant are non-deterministic, there may arise blocking states, even though the supervisor itself is nonblocking and the plant is nonmarking. If the closed-loop system is to be nonblocking, we must guarantee that these states are either not accessible, or marked. If such states are marked, they are, by definition, nonblocking; the system is allowed to rest in these states indefinitely.

To be able to always determine when the closed-loop system will be trace nonblocking, we make the following definition.

**Definition 3.24 Deterministically Marked**

A supervisor $S$ is said to be *deterministically marked* with respect to a plant $P$, if for all strings present in both languages and for all states reached by such a string in the plant and in the supervisor, a supervisor state is marked whenever its ready set is disjoint from the ready set of at least one of these plant states. That is,

$$\begin{aligned} &S \text{ deterministically marked} \\ &\Leftrightarrow \forall s \in L(P \parallel S)\, \forall p \in \delta(P, s)\, \forall q \in \delta(S, s)\, out(p) \bigcap out(q) = \emptyset \rightarrow q \in M_S \end{aligned} \tag{3.56}$$

**Remark.** This means that for $S$ to be deterministically marked with respect to $P$, all terminating states of $S$, and all states corresponding to terminating states of $P$ must be marked. Also, if a state of $S$ and a state of $P$, reached by the same string, have disjoint ready sets, then this state must be marked. This corresponds to states that will be terminating in the closed-loop system.

Since we always assume that $P$ is nonmarked, the expression above can be equivalently rewritten as

$$\forall s \in L(P \parallel S)\, \forall \langle p, q \rangle \in \delta(P \parallel S, s)\ \mathrm{out}(\langle p, q \rangle) \neq \emptyset \vee \langle p, q \rangle \in M_{P \parallel S}, \tag{3.57}$$

which emphasizes the fact that when $S$ is deterministically marked with respect to $P$, then terminating states of the closed-loop system are marked. Interpreting marking as "allowed by the specification to be terminating ", then $S$ deterministically marked means that the closed-loop system is "nonblocking" in the sense of Overkamp (1994).

When $P$ is deterministic, $\delta(P, s)$ is singular. When $L(S) \subseteq L(P)$ we have that the ready set of any state $q = \delta(S, s)$, is a subset of the ready set of the state $p = \delta(P, s)$, so

**Example 3.5 Deterministically Marked Transition Machine**

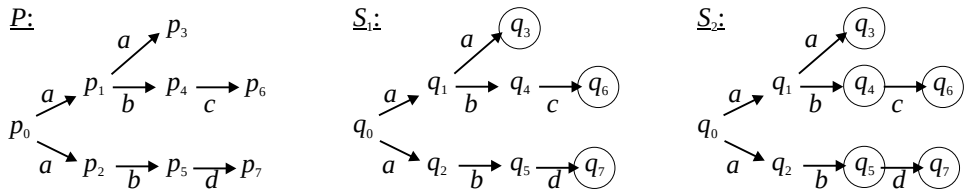



**Figure 3.9:** A plant and two trace nonblocking supervisors. The supervisor $S_1$ is not deterministically marked, while $S_2$ is.

In Figure 3.9 is shown a plant $P$ and two supervisors $S_1$ and $S_2$. Both of these are trace nonblocking. However, $S_1$ is not deterministically marked, since $\delta(P, ab) = \{p_4, p_5\}$, $\delta(S_1, ab) = \{q_4, q_5\}$, and $\text{out}(p_4) \bigcap \text{out}(q_5) = \emptyset$ as well as $\text{out}(p_5) \bigcap \text{out}(q_4) = \emptyset$, but neither $q_4$ nor $q_5$ are marked. Of course, marking these states generates a deterministically supervisor, as $S_2$.

■

that for all $q = \delta(S, s)$ $\text{out}(p) \bigcap \text{out}(q) = \text{out}(q)$. Therefore, when $P$ is deterministic and $L(S) \subseteq L(P)$, Definition 3.24 is equivalent to

$$\forall s \in L(P \| S)\, \forall q \in \delta(S, s)\, \text{out}(q) \neq \emptyset \vee q \in M_S. \tag{3.58}$$

That is, for a deterministic $P$, when $L(S) \subseteq L(P)$, $S$ is deterministically marked, if and only if all terminating states of $S$ are marked. □

For a deterministically marked supervisor, not only are all terminating states marked so that the corresponding terminating states of the closed-loop system are also marked, but all states that become terminating states in the closed-loop system, are also marked. Thus, all terminating states of the closed-loop system are marked. In addition, any other state that is marked in the supervisor and survives under synchronization with the plant will also be marked in the closed-loop system. Thus, the closed-loop system is trace nonblocking, if the supervisor is trace nonblocking and deterministically marked. This is shown in Lemma 3.26 below.

But first we must show the following important equivalence, which will be used in the following proofs.

**Lemma 3.25** For two states $p$ and $q$ of two transition machines $P$ and $S$, with $\Sigma_S = \Sigma_P$, the following equivalence holds,

$$\text{out}(p) \bigcap \text{out}(q) \neq \emptyset \Leftrightarrow \exists s' \in \Sigma_P^+\, \delta_P(p, s') \neq \emptyset \wedge \delta_S(q, s') \neq \emptyset, \tag{3.59}$$

where $\Sigma_P^+$ is the set of all nonempty finite strings over $\Sigma_P$.

**Proof.** ($\Rightarrow$) When $\text{out}(p) \bigcap \text{out}(q) \neq \emptyset$ there exists an event $\sigma \in \Sigma_P$ such that $\sigma \in \text{out}(p) \bigcap \text{out}(q)$. Thus, $\delta_P(p, \sigma) \neq \emptyset$ and $\delta_S(q, \sigma) \neq \emptyset$. Since $\Sigma_P \subseteq \Sigma_P^+$, $s'$ exists and can be chosen equal to $\sigma$.

($\Leftarrow$) When there exists a nonempty string $s' \in \Sigma_P^+$ such that $\delta_P(p, s') \neq \emptyset$ and $\delta_S(q, s') \neq \emptyset$, then there exists an event $\sigma \in \Sigma_P$ such that $s' = \sigma s''$. Obviously, $\sigma \in \mathrm{out}(p) \bigcap \mathrm{out}(q)$. ■

**Remark.** Note that the implication

$$\mathrm{out}(p) \bigcap \mathrm{out}(q) \neq \emptyset \Rightarrow \exists s' \in \Sigma_P^* \; \delta_P(p, s') \neq \emptyset \wedge \delta_S(q, s') \neq \emptyset \tag{3.60}$$

is also proved by this lemma. Since $s'$ may then be chosen to be the null string, it is not necessarily so that a common event exists in the ready sets. Furthermore, the implication

$$\exists s' \in \Sigma_P^+ \; \delta_P(p, s') \neq \emptyset \wedge \delta_S(q, s') \bigcap A \neq \emptyset \Rightarrow \mathrm{out}(p) \bigcap \mathrm{out}(q) \neq \emptyset \tag{3.61}$$

for $A \subseteq Q_S$ follows readily.

□

Now we are equipped with tools to prove the following lemmas.

**Lemma 3.26** For a plant $P$ and a supervisor $S$, the closed-loop system will be trace nonblocking if the supervisor is trace nonblocking and deterministically marked.

**Proof.** When $S$ is deterministically marked we have by (3.57)

$$\forall s \in L(P \,\|\, S) \; \forall \langle p, q \rangle \in \delta(P \,\|\, S, s) \; \mathrm{out}(\langle p, q \rangle) \neq \emptyset \vee \langle p, q \rangle \in M_{P \,\|\, S}, \tag{3.62}$$

which, by (3.60) implies that

$$\forall s \in L(P \,\|\, S) \; \forall \langle p, q \rangle \in \delta(P \,\|\, S, s) \; \exists s' \in \Sigma_P^* \; \delta_{P \,\|\, S}(\langle p, q \rangle, s') \neq \emptyset \vee \langle p, q \rangle \in M_{P \,\|\, S}, \tag{3.63}$$

using that $\mathrm{out}(\langle p, q \rangle) = \mathrm{out}(p) \bigcap \mathrm{out}(q)$. When $S$ is trace nonblocking, we have by (3.51)

$$\forall s \in L(S) \; \forall q \in \delta(S, s) \; \exists s' \in \Sigma_S^* \; \delta_S(q, s') \bigcap M_S \neq \emptyset. \tag{3.64}$$

Since this holds for all strings of $L(S)$, it specifically holds for all strings of $L(P \,\|\, S)$.

We have that $\delta_{P \,\|\, S}(\langle p, q \rangle, s) \neq \emptyset \Leftrightarrow \delta_P(p, s) \neq \emptyset \wedge \delta_S(q, s) \neq \emptyset$. Since $S$ is trace nonblocking there always exist a string $s'$ so that $\delta_S(q, s') \bigcap M_S \neq \emptyset$. If this string is not defined from $p$, so that $\delta_P(p, s') = \emptyset$, then $\delta_{P \,\|\, S}(\langle p, q \rangle, s') = \emptyset$ so that $\langle p, q \rangle$ is marked, since $S$ is deterministically marked. If, on the other hand, $\delta_P(p, s') \neq \emptyset$, then, since $S$ is nonblocking, $\delta_{P \,\|\, S}(\langle p, q \rangle, s') \bigcap M_{P \,\|\, S}$. This follows from the fact that $M_{P \,\|\, S} = Q_P \times M_S$.

Thus, when $S$ is both deterministically marked and trace nonblocking, the closed-loop system will be trace nonblocking. ■

**Remark.** In practice, it is not required that the supervisor is entirely trace nonblocking, as defined by Definition 3.23. It is in fact only required that the supervisor is trace nonblocking with regard to the *strings arising in the closed-loop system.* □

We can also show that, whenever the closed-loop system is trace nonblocking, the supervisor is also deterministically marked. This is shown by the following lemma.

**Lemma 3.27** For a plant $P$ and a supervisor $S$, the supervisor is deterministically marked if $P \,\|\, S$ is trace nonblocking.

**Proof.** By (3.53) the closed-loop system is trace nonblocking if and only if

$$\forall s \in L(P \,\|\, S)\, \forall \langle p, q\rangle \in \delta(P \,\|\, S, s)\, \exists s' \in \Sigma_P^*\, \delta_{P\,\|S}(\langle p, q\rangle\,, s') \bigcap M_{P\,\|S} \neq \emptyset. \tag{3.65}$$

Since $\Sigma_P^* = \Sigma_P^+ \bigcup \{\varepsilon\}$, this expression can be equivalently rewritten as

$$\begin{aligned} \forall s \in L(P \,\|\, S)\, \forall \langle p, q\rangle \in \delta(P \,\|\, S, s)\, \exists s' \in \Sigma_P^+ \; \Big[\delta_{P\,\|S}(\langle p, q\rangle\,, s') \bigcap M_{P\,\|S} \neq \emptyset\Big] \vee \\ \Big[\delta_{P\,\|S}(\langle p, q\rangle\,, \varepsilon) \bigcap M_{P\,\|S} = \langle p, q\rangle\Big] \end{aligned} \tag{3.66}$$

where $\delta_{P\,\|S}(\langle p, q\rangle\,, \varepsilon) = \langle p, q\rangle$ by definiton. For $P\| S$ to be trace nonblocking, either $\langle p, q\rangle$ is marked, or some marked state is reachable by $s'$.

Using Lemma 3.25 and the fact that $\mathrm{out}(p) \bigcap \mathrm{out}(q) = \mathrm{out}(\langle p, q\rangle)$, we can now write the implied expression

$$\forall s \in L(P \,\|\, S)\, \forall \langle p, q\rangle \in \delta(P \,\|\, S, s)\, \mathrm{out}(\langle p, q\rangle) \neq \emptyset \vee \langle p, q\rangle \in M_{P\,\|S} \tag{3.67}$$

which clearly shows that $S$ is deterministically marked, see (3.57). ■

We cannot show, however, that when the closed-loop system is trace nonblocking so will also $S$ be. Naturally, this is a consequence of the definition of trace nonblocking, regarding all strings in the language $L(S)$. The closed-loop system can only tell about the subset of $L(S)$ defined by $L(P \,\|\, S)$. If $L(S) \subseteq L(P)$, though, it can be shown that $S$ is trace nonblocking if the closed-loop system is.

**Lemma 3.28** For a plant $P$ and a supervisor $S$, with $L(S) \subseteq L(P)$, the supervisor is trace nonblocking if the closed-loop system $P \,\|\, S$ is trace nonblocking.

**Proof.** The only thing that prohibited us from claiming that (3.53) implies (3.51) was the fact that the latter concerns $L(S)$, while the former concerns $L(P \,\|\, S) \subseteq L(S)$, in general. However, when $L(S) \subseteq L(P)$ we have that $L(P \,\|\, S) = L(S)$, so that we can exchange one for the other as we please. Under this assumption it should be clear that (3.53) implies (3.51). ■

Finally, we arrive at the following, by now obvious, theorem.

**Theorem 3.29** For a plant $P$ and a supervisor $S$, with $L(S) \subseteq L(P)$, the closed-loop system $P \,\|\, S$ is trace nonblocking if and only if $S$ is trace nonblocking and deterministically marked.

**Proof.** ( $\Rightarrow$) By Lemma 3.27 and Lemma 3.28 we know that when $L(S) \subseteq L(P)$ and $P \,\|\, S$ is trace nonblocking, then $S$ is trace nonblocking and deterministically marked.

( $\Leftarrow$) By Lemma 3.26 we know that when $S$ is trace nonblocking and deterministically marked, then $P \,\|\, S$ is trace nonblocking. ■

**Remark.** Note that, in general, $P \| S$ does not become deterministically marked with respect to $P$. Thus, the system $P \| (P \| S)$ is not necessarily trace nonblocking, even though both $S$ and $P \| S$ may be. This means that $P \| S$ cannot be used as a supervisor for $P$ to generate a nonblocking closed-loop system; a strategy that is sometimes employed for deterministic systems. Furthermore, when $P$ is non-deterministic, the system $P \| (P \| S)$ is not necessarily equal to $P \| S$; something which is sometimes relied upon for deterministic systems. See Kumar (1991), for instance. □

By Theorem 3.29 we know that the closed-loop system of plant and supervisor $P \| S$ is nonblocking if and only if $S$ is nonblocking and deterministically marked. By the remark to Definition 2.24 we know that a necessary condition for $S$ to be nonblocking is that all its reachable terminating states are marked. In the remark to Definition 3.24 it was shown that when $P$ is deterministic and $L(S) \subseteq L(P)$, then $S$ is deterministically marked if and only if all terminating states are marked. Thus, when $P$ is deterministic and $L(S) \subseteq L(P)$, the closed-loop system is nonblocking if and only if $S$ is nonblocking. This is emphasized in the following corollary to Theorem 3.29.

**Corollary 3.30** For a deterministic plant $P$ and a supervisor $S$, with $L(S) \subseteq L(P)$, the closed-loop system $P \| S$ is trace nonblocking if and only if $S$ is trace nonblocking.

In the final section of this we will use the lemmas and theorems defined in the previous sections, to show existence of a complete supervisor for a nonblocking closed-loop system under various combinations of deterministic and/or non-deterministic plant, specification and supervisor. This is the main contributions of this thesis.

## 3.5 Some Notes on Non-deterministic Supervisory Control

In this section we will finally tie together the aspects of non-deterministic supervisory control that have been described in the previous chapters. We will examine combinations of deterministic and non-deterministic plant $P$, specification $Sp$, and supervisor $S$, and give sufficient and necessary conditions for the existence of a complete supervisor such that the closed-loop system is nonblocking. Note that when we speak of a *deterministic specification*, we will regard the language $L(Sp)$. Thus, we assume that a deterministic specification imposes a restriction on the language of the closed-loop system, whether this system is deterministic or not. We will regard the plant as nonmarking, that is $L_m(P) = L(P)$, so that only the specification and the supervisor introduces marking.

There are other interpretations of the interconnection of plant and supervisor, such as the inclusion of *driven events* described by Kumar (1994) and Shayman (1994). There are also other interpretations of how the specification is to influence the behavior of the closed loop system, such as the notion of *reduction*, described by Overkamp (1994) and Overkamp (1995).

**Theorem 3.31** For a deterministic plant $P$, and a specification language $L(Sp)$, there exists a deterministic, complete supervisor such that $L_m(P \| S) \subseteq L_m(Sp)$, $L(P \| S) \subseteq$

$L(Sp)$ and $\overline{L_m(P \,\|\, S)} = L(P \,\|\, S)$, if and only if there exists a controllable prefix closed sublanguage $K \subseteq L(P) \bigcap L(Sp)$ and a marked language $K_m \subseteq L(P) \bigcap L_m(Sp)$, such that $K = \overline{K_m}$.

**Proof.** ($\Rightarrow$) Choose $K = L(P \,\|\, S)$. When $S$ is complete, $L(P \,\|\, S) = K$ is controllable. Since $L(P \,\|\, S)$ is always a sublanguage of $L(P)$, $L(P \,\|\, S) \subseteq L(Sp)$ means that $L(P \,\|\, S) = K \subseteq L(P) \bigcap L(Sp)$. Choose $K_m = L_m(P \,\|\, S)$. Since $L_m(P \,\|\, S) \subseteq L(P) \bigcap L_m(S)$, $L_m(P \,\|\, S) \subseteq L_m(Sp)$ means that $L_m(P \,\|\, S) \subseteq L(P) \bigcap L_m(Sp)$, so that $K_m \subseteq L(P) \bigcap L_m(Sp)$. When $\overline{L_m(P \,\|\, S)} = L(P \,\|\, S)$, then $\overline{K_m} = K$.

($\Leftarrow$) Choose $S$ such that $L_m(S) = K_m$ and $L(S) = K$. This can always be achieved since $K_m \subseteq K$. Since $K$ is controllable, $S$ is complete. Since $L(S) = K \subseteq L(P) \bigcap L(Sp)$, $L(P \,\|\, S) = L(P) \bigcap L(S) = L(S) \subseteq L(P) \bigcap L(Sp) \subseteq L(Sp)$. Since $L_m(S) = K_m \subseteq L(P) \bigcap L_m(Sp)$, $L_m(P \,\|\, S) = L(P) \bigcap L_m(S) = L_m(S) \subseteq L(P) \bigcap L_m(Sp)$. When $\overline{K_m} = K$, then $\overline{L_m(S)} = L(S)$ so that $\overline{L_m(P \,\|\, S)} = L(P \,\|\, S)$. ■

**Remark.** This is, of course, the original supervisory control problem proposed by Ramadge (1987) and Wonham (1987). If $K$ is the supremal controllable sublanguage of $L(P) \bigcap L(Sp)$, then $S$ is also minimally restrictive. Since we choose $L(S) = K \subseteq L(P)$, and both $S$ and $P$ are deterministic, we know that $S$ is also inverse complete. □

**Theorem 3.32** For a deterministic plant $P$, and a specification language $L(Sp)$, there exists a non-deterministic, complete supervisor such that $L_m(P \,\|\, S) \subseteq L_m(Sp)$, $L(P \,\|\, S) \subseteq L(Sp)$ and $\overline{L_m(P \,\|\, S)} = L(P \,\|\, S)$, if and only if there exists a controllable prefix closed sublanguage $K \subseteq L(P) \bigcap L(Sp)$ and a marked language $K_m \subseteq L(P) \bigcap L_m(Sp)$, such that $K = \overline{K_m}$.

**Remark.** Obviously, this is the same as Theorem 3.31 above. Since we only regard the languages of the closed-loop system and the specification, we can choose whatever non-deterministic supervisor we want, as long as its language satisfies the conditions given in Theorem 3.31 above. Note though, that given a controllable language, the chosen supervisor with this language has to be conforming to be complete.

Theorem 3.32 is essentially the problem studied by Inan (1994). Given a controllable sublanguage of $L(P)$, a deterministic marked state-machine specification with this language is generated. Any such state-machine is applicable, and it is, of course, complete. From this specification is then generated a deterministic supervisor, essentially by synchronizing the plant with the specification. However, under the assumption that not all plant events are observable, the deterministic supervisor is augmented so as not to rely on observations of unobservable events. This augmentation generates a non-deterministic supervisor, which is then pruned to be nonblocking.

Note though that, given this controllable language $K$, when choosing a non-deterministic supervisor $S$ such that $L(S) = K$, we have in Section 3.4 shown that $S$ must be conforming with respect to the plant. Of course, a deterministic $S$ is always conforming. □

**Theorem 3.33** For a deterministic plant $P$, and a non-deterministic specification $Sp$, there exists a complete supervisor $S$ such that $P \,\|\, S \leq Sp$ and $\overline{tr_m(P \,\|\, S)} = tr(P \,\|\, S)$,

if and only if there exists a complete subprocess $S' \leq Sp$ such that $S'$ refines $P$ and $S'$ is trace nonblocking.

**Proof.** ($\Rightarrow$) Choose $S' = P \| S$. Since $S$ is complete and $P$ is conforming with respect to $\Sigma_u$, by Theorem 3.19, $S'$ is complete. Since the closed-loop system is trace nonblocking, $S'$ is trace nonblocking. Since $P \| S \leq Sp$, we have that $S' \leq Sp$. From Theorem 2.48 it follows that $S' = P \| S$ always refines $P$.

($\Leftarrow$) Choose $S = S' \leq Sp$. Since $S$ refines $P$, and $P$ is deterministic, it follows from Theorem 2.48 that $P \| S = S \leq Sp$. Since $S$ is trace nonblocking and $P$ is deterministic, by Corollary 3.30 we know that $P \| S$ is trace nonblocking. ■

**Remark.** If $S'$ is the largest complete subprocess of $Sp$ refining $P$, then $S$ is also minimally restrictive. Since we choose $S = S'$ which refines $P$, so that $L(S) \subseteq L(P)$, and since $P$ is deterministic and therefore a conforming plant, we know that $S$ is also inverse complete.

This is the problem that initially launched the work that has resulted in this thesis. As described by the introductory example, when applying object oriented-modeling principles to complex manufacturing systems, a non-deterministic specification arises naturally by the interleaving of all the desired product routes. From such a specification, a non-deterministic supervisor is generated; see the next chapters. □

**Theorem 3.34** For a non-deterministic plant $P$, and a specification language $L(Sp)$, there exists a deterministic, complete supervisor such that $L_m(P \| S) \subseteq L_m(Sp)$, $L(P \| S) \subseteq L(Sp)$ and $\overline{L_m(P \| S)} = L(P \| S)$, if and only if there exists a controllable prefix closed sublanguage $K \subseteq L(P) \bigcap L(Sp)$ and a marked language $K_m \subseteq L(P) \bigcap L_m(Sp)$, such that $K = \overline{K_m}$.

**Remark.** This is close to the problem studied by Kumar (1994) and Shayman (1994), though take a trajectory-model point of view with the inclusion of driven events. Driven events can be refused by the plant, and this is an essential feature of the successful control design. Driven events are therefore *not* equivalent to the controllable command events of Balemi (1992). The closed-loop system is *trajectory model nonblocking*, a weaker requirement than trace nonblocking, but stronger than language nonblocking.

Since the specification is given as a language, the necessary and sufficient conditions for existence of a supervisor are the same as given in Theorem 3.31. This is also noted by Shayman (1994), but it only holds when there are no driven events. Otherwise $K$ is required to be controllable with respect to the *augmented* plant, which depends on the trajectory model of $P$. The augmented plant selfloops at every state on the driven events not defined by the ready set. Note also that non-determinism is modeled by silent transitions. □

In addition we note that Overkamp (1994) and Overkamp (1995) have studied the combination of non-deterministic plant and specification. For this problem a deterministic supervisor is generated. In Overkamp (1994) automata are used for the supervisor synthesis, while in Overkamp (1995), the failures model is used. Furthermore, Overkamp

requires the closed-loop system to *reduce* the specification; that is, $L(P \parallel S) \subseteq L(Sp)$ and $P \parallel S$ does not reach a terminating state after a string $s$, unless $Sp$ also does. Reduction is related to the notion of a deterministically marked supervisor, as mentioned in the remark to Definition 3.24. Note though that marking as such is not treated by Overkamp.

## 3.6 Chapter Summary

In this chapter we have extended the supervisory control theory to the case of non-deterministic specification. We have also discussed and shown some results for the case where the plant is non-deterministic as well. The basic supervisory control theory have been presented in a unified framework, with the synchronous composition as the means to disable events of the plant. Necessary and sufficient conditions for the existence of a supervisor have been given for some combinations of non-deterministic plant, specification and supervisor. The main result being, of course, the existence of a non-deterministic supervisor given a deterministic plant and a non-deterministic specification, Theorem 3.33.

The results of Section 3.2 and Section 3.3 are not new. The synchronous composition as a means for the supervisor to control the plant was introduced already by Kumar (1991), albeit without considering marking. The *derivation* of the controllability property is, as far as we know, our own. Wonham (1987), Ramadge (1987), Balemi (1992) and Kumar (1995), among others, *define* controllability but do not *derive* it.

The results of Section 3.4 are generalizations of known results. The definition of conformity, Definition 3.15, is our own, and it is a relaxation of the requirement of deterministic processes given by many authors. Ramadge (1987) and Kumar (1995), among others. The necessary and sufficient condition for choosing a non-deterministic supervisor given a controllable language, Theorem 3.18, is a new result.

The results obtained for the input/output interpretation of Balemi (1992) are also generalizations of known results. The inverse properties have been generalized by us, but the original definitions come from Balemi (1992).

The definition of a deterministically marked supervisor, Definition 3.24, is new. So is also the necessary and sufficient condition for when the closed-loop system of a non-deterministic plant and a non-deterministic supervisor is trace-nonblocking, Theorem 3.29. The results shown in Section 3.5 are our own only as concerns Theorem 3.33. The other results have not been collected and presented in such a unified framework before, though.

# Chapter 4

# Supervisor Synthesis

For supervisor synthesis we need efficient algorithms for calculating the complete and trim subprocess of a given marked supervisor "candidate" $S$, given a non-marked plant $P$. Naturally, there may exist several solutions. The null process, for instance, is always a solution since it is both complete and trim. However, an additional requirement is to find the *supremal* complete and trim subprocess of $S$.

The problem will be broken down into finding the supremal complete and accessible subprocess and the supremal coaccessible subprocess of $S$, respectively. First we will focus on finding the supremal accessible, the supremal coaccessible and the supremal trim, that is, accessible and coaccessible, subprocess of a given process. Then we use the algorithm for finding the supremal accessible subprocess, when we calculate the supremal accessible and complete subprocess. Finally, we use those algorithms to find the supremal complete *and* trim subprocess.

In this chapter we observe certain properties of the characterizations of accessibility, coaccessibility and completeness. For each characterization, we give general conditions under which we can find the supremal subprocesses holding the given property, under some additional constraints. We also give general fixpoint algorithms for finding these subprocesses. Finally, we specialize these algorithms to the specific problems of finding the supremal trim, and the supremal trim and complete subprocesses of $S$.

## 4.1 Supremal Elements

The number of subprocesses of a given process, $P$, is finite and limited by the number of possible subsets of $E_P$ and $Q_P$. The set of all subprocesses of $P$, $\mathcal{S}(P)$ of Definition 2.34 has been shown to be a partial ordering, by Theorem 2.33. The fact is, that given the union and intersection operations on subprocesses, defined in Definition 2.50, the set of all subprocesses is a *lattice* with $P$ itself as the least upper bound and $\emptyset_P$ as the greatest lower bound.

> *A* lattice *is a partially ordered set* $\langle L, \leq \rangle$ *in which every pair of elements* $a, b \in L$ *has a greatest lower bound and a least upper bound.*
>
> Tremblay (1987), Definition 4-1.1

Here, $\langle L, \leq \rangle$ is an ordered pair of a set $L$ and its ordering relation $\leq$.

When working on the problem to find a proper supervisor among the subprocesses of the supervisor candidate, we can make use of properties pertaining to lattices. We will view lattices as algebraic systems, since then many concepts that are associated with algebraic systems can also be applied to lattices. Thus, it will be possible to define semilattices, for instance. The following is a reformulation of Definition 4-1.2, given by Tremblay (1987), who also shows that this definition is equivalent to the Definition 4-1.1 cited above.

**Definition 4.1 Lattice**

A lattice is an algebraic system $\langle L, \bigcup, \bigcap \rangle$ with two binary operations $\bigcup$ and $\bigcap$ on $L$ that are both are both associative, commutative and satisfy the absorption laws (4.1),

$$a \bigcup (a \bigcap b) = a \text{ and } a \bigcap (a \bigcup b) = a. \tag{4.1}$$

for any $a, b \in L$.

**Remark.** This has the consequence that under the $\bigcup$and $\bigcap$ operators all elements of $L$ are idempotent. That is, for $a \in L$ we have that $a \bigcup a = a \bigcap a = a$. It also follows that a lattice is closed under the operations and $\bigcup$ and $\bigcap$, that is, for $a, b \in L$ we have that $a \bigcup b \in L$ and $a \bigcap b \in L$. This is a property of a binary operation, see Tremblay (1987). □

Every subset of a partially ordered set is still a partially ordered set under the same partial ordering relationship, but not all subsets of a lattice are necessarily lattices. That is, for a lattice $\langle L, \leq \rangle \Leftrightarrow \langle L, \bigcup, \bigcap \rangle$, a (finite) subset $X \subseteq L$ is a partial ordering $\langle X, \leq \rangle$, but $\langle X, \bigcup, \bigcap \rangle$ is not necessarily closed under the binary operations $\bigcup$ and $\bigcap$. However, $\langle X, \bigcup, \bigcap \rangle$ may be closed under $\bigcup$, in which case it is said to be an *upper semilattice*, or it may be closed under $\bigcap$ and is then a *lower semilattice*.

**Definition 4.2 Upper and Lower Semilattices**

Let $X \subseteq L$ be a subset of the set $L$ of the lattice $\langle L, \bigcup, \bigcap \rangle$. The partial ordering $\langle X, \leq \rangle$ is an *upper semilattice* if $X$ is closed under $\bigcup$ , and $\langle X, \leq \rangle$ is a *lower semilattice* if $X$ is closed under $\bigcap$.

**Remark.** When $\langle X, \leq \rangle$ is an upper semilattice, for all $x_1, x_2 \in X$ we have that $x_1 \bigcup x_2 \in X$, but it may be that $x_1 \bigcap x_2 \notin X$. Similarly for lower a semilattice, $x_1 \bigcup x_2$ may not be in the set. See also Example 4.1. When $\langle X, \leq \rangle$ is both an upper and a lower semilattice under the operations $\bigcup$ and $\bigcap$, it is a *sublattice* of $\langle L, \bigcup, \bigcap \rangle$.

Note also that $\langle X, \leq \rangle$ may be a lattice even though it is not closed under either of the operations $\bigcup$ and $\bigcap$. $\langle X, \leq \rangle$ may be closed under two binary operations, say $\oplus$ and $\otimes$, different from $\bigcup$ and $\bigcap$, respectively. It is not a sublattice of $\langle L, \bigcup, \bigcap \rangle$, then. □

It is well-known, see Tremblay (1987), that for any (finite) subset $X \subseteq L$ there exist a unique *least upper bound* or *supremum*, and a unique *greatest lower bound* or *infimum*. These can be expressed as

$$\sup X = \bigcup_{x_i \in X} x_i \text{ and } \inf X = \bigcap_{x_i \in X} x_i. \tag{4.2}$$

**Example 4.1 Partial Orderings, Lattices and Semilattices**

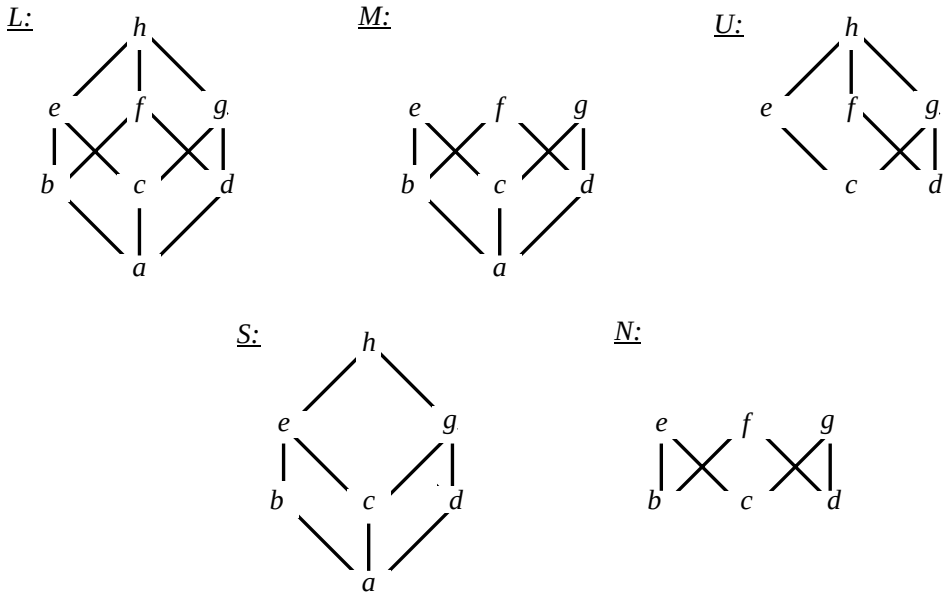



**Figure 4.1:** Examples of lattices, semilattices and partially ordered sets.

In Figure 4.1 is shown the Hasse diagrams of a number of partially ordered sets. The ordering is implied by the lines between the set elements. The transitiveness of a partial ordering is employed to not draw an excessive amount of lines. For instance, the elements $e$ and $c$, that are present in all sets, are ordered as $c \leq e$. Since $a$ is ordered as $a \leq c$ in $L$, for instance, this also implies that $a \leq e$. We can let the lines denote the binary operations on the elements of the sets. For $\bigcup$ we move upwards, and for $\bigcap$ downwards. For instance, in $L$ $b \bigcup d = f$ and $b \bigcap d = a$.

The partially ordered set $L$ is a lattice. It is closed both under $\bigcup$ and $\bigcap$. All other sets are subsets of $L$. The set $M$ is not a lattice, but a lower semilattice, since $\sup(M) \notin M$, that is $e \bigcup f = e \bigcup g = f \bigcup g = h \notin M$. However, $\inf(M) = a \in M$. Similarly, $U$ is an upper semilattice, since $\sup(U) = h \in U$. $N$ is not a lattice since neither $\sup(N) = h$ nor $\inf(N) = a$ belong to $N$. $S$ is a subset of $L$ forming a lattice. Note that the binary operation $\bigcup$ is not the same operation for $S$ as for $L$. In $L$, $b \bigcup d = f$, while in $S$ $b \bigcup d = h$. Thus, $S$ is not a sublattice of $L$.

■

However, it is not necessarily so that $\inf(X) \in X$ and/or $\sup(X) \in X$, though it will always hold that $\inf(X) \in L$ and $\sup(X) \in L$. It is also not necessarily so that both $\sup(X)$ and $\inf(X)$ belong to $X$ simultaneously. When are we guaranteed that $\sup(X) \in X$ and $\inf(X) \in X$, for $X \subseteq L$? We can show the following lemma, which follows readily from the definitions of sup and inf, and therefore we will give no proof.

**Lemma 4.3** For a lattice $\langle L, \bigcup, \bigcap \rangle$, a (finite) subset $X \subseteq L$ contains its least upper bound and its greatest lower bound, if $X$ is closed under the binary operations $\bigcup$ and $\bigcap$. That is,

$$\begin{aligned} &\forall x_1, x_2 \in X \; x_1 \bigcup x_2 \in X \Rightarrow \sup X \in X \quad \text{and} \\ &\forall x_1, x_2 \in X \; x_1 \bigcap x_2 \in X \Rightarrow \inf X \in X \end{aligned} \tag{4.3}$$

**Remark.** Note though that $X$ being closed under $\bigcup$ (and $\bigcap$) is a *sufficient* condition. The least upper bound of $X$ can belong to $X$, without $X$ being closed under $\bigcup$. A necessary and sufficient condition for $\sup X \in X$ is in fact

$$\forall x_1, x_2 \in X \; \exists x_3 \in X \; x_1 \bigcup x_2 \leq x_3. \tag{4.4}$$

However, this is of no importance for the following presentation. □

Subprocess union and subprocess intersection are in essence defined as ordinary set union and set intersection on ordered 5-tuples of sets, see Definitions 2.50. Of course, set union and set intersection hold the properties required above for lattices, and thus, it is obvious that the set of all subprocesses, $\mathcal{S}(P)$, of any transition machine $P$, is indeed a lattice.

Subsets of $\mathcal{S}(P)$ can be distinguished by demanding certain structural properties of the elements, *characterizations*. Typically such characterizations concern accessibility, coaccessibility and completeness. The subsets of $\mathcal{S}(P)$ thus distinguished consist of all subprocesses of $P$ holding the requested property. This notion can be generalized as in the following definition.

**Definition 4.4 Set of All Characterized Subprocesses**

Given a characterization, $\mathcal{X}(\cdot)$, of subprocesses of a process $P$, $\mathcal{X}(P) \subseteq \mathcal{S}(P)$ is the set of all subprocesses of $P$ satisfying $\mathcal{X}(\cdot)$. That is,

$$\mathcal{X}(P) = \{ S \leq P | \, S \;\; \text{satisfies } \mathcal{X}(\cdot) \}. \tag{4.5}$$

**Remark.** $P$ is called the *generating element*, since from the characterization $\mathcal{X}(\cdot)$ it generates the set $\mathcal{X}(P)$. Of course, a characterized set is a subset of the set of all subprocesses of the generating element. Therefore, each characterized set is partially ordered. Some characterizations though, are such that the generated set is not closed under subprocess union and/or intersection. When a characterized subset is closed under subprocess union it will be said to be *union closed*. When it is closed under both union and intersection it will be said to be *closed*. The characterized subset is then a sublattice of $\mathcal{S}(P)$.

The closedness of a characterization is an inherent feature of the characterization itself, not of the generating element. Thus, for a union closed characterization its generated set will always be an upper semilattice, no matter what the generating element looks like. It may happen that the generated set is empty, but then we trivially have a lattice.

Characterizations like $\mathcal{X}(\cdot)$, are normally called *predicates*, see Tremblay (1987), but since we will use $\mathcal{X}(P)$ to denote the set of all elements of $\mathcal{S}(P)$ satisfying $\mathcal{X}(\cdot)$, we have chosen to speak of *characterizations*. When regarding $\mathcal{X}(\cdot)$ as a predicate, $\mathcal{X}(P)$ is either true or false, so that the generated set would have to be given as $\{P' \in \mathcal{S}(P) \,|\mathcal{X}(P')\}$, the *extension* of $\mathcal{X}(\cdot)$ in the set $\mathcal{S}(P)$. An element $P''$ belongs to this set if and only if $\mathcal{X}(P'')$ is true. Viewing $\mathcal{X}(\cdot)$ as a characterization and thus having $\mathcal{X}(P)$ denote the generated set, gives us a shorter and more concise notation. □

Of specific interest to us will be characterizations such that they are closed under the operation of subprocess union. Such characterizations define upper semilattices of $\mathcal{S}(P)$. As shown by Lemma 4.3, for such characterizations there exists a unique supremal element satisfying the characterization, and of which all other elements satisfying the same characterization are subprocesses. This element is obtained by taking the union of all elements satisfying the characterization. The supremal element of the characterized upper semilattice $\mathcal{X}(P)$ thus "contains" all subprocesses of $P$ that satisfy the characterization $\mathcal{X}(\cdot)$; that is all elements of $\mathcal{X}(P)$. Naturally, for $\mathcal{S}(P)$, the supremal element is $P$ itself. Likewise, the inifimal element of $\mathcal{S}(P)$ is obtained by intersecting all subprocesses of $P$, which will of course yield the null process, $\emptyset_P$.

In this section we are set out to show that given an arbitrary process $P$ and two union closed characterizations, $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$, we can *efficiently* find the supremal element satisfying both characterizations. By *efficiently* we mean, without using the brute force method of calculating the entire sets $\mathcal{X}(P)$ and $\mathcal{Y}(P)$, intersecting them and then calculating the union over all elements of the intersection. The brute force method would naturally give a correct result, however, in practice it would be unusable due to the large amount of subprocesses that would have to be generated. An efficient method would only generate those subprocesses of $P$ that were absolutely necessary in order to guarantee that the sought element was found. Such an efficient method does exist, as will be shown.

To prove the claim above, we will need the following "toolbox".

**Lemma 4.5** With two union closed characterizations $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$, and generating elements $P', P'' \leq P$ for the generated sets $\mathcal{X}(P), \mathcal{X}(P'), \mathcal{Y}(P'') \subseteq \mathcal{S}(P)$, the following holds

1. $\mathcal{X}(P') \subseteq \mathcal{Y}(P'') \Rightarrow \sup \mathcal{X}(P') \leq \sup \mathcal{Y}(P'')$
2. $\sup \mathcal{X}(P') = P' \Leftrightarrow P' \in \mathcal{X}(P')$
3. $\mathcal{X}(P) \subseteq \mathcal{S}(\sup \mathcal{X}(P))$
4. $\mathcal{X}(P') = \mathcal{X}(\sup \mathcal{X}(P'))$
5. $\mathcal{X}(P') = \mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{X}(P'))$
6. $\mathcal{X}(P') = \mathcal{X}(P) \bigcap \mathcal{S}(P')$

**Remark.**

1. If all elements of $\mathcal{X}(P')$ are also elements of $\mathcal{Y}(P'')$, then the union over all elements of $\mathcal{X}(P')$ cannot be "larger" than the union over all elements of $\mathcal{Y}(P'')$.

2. If the supremal element of the generated set is equal to the generating element, then this element satisfies the characterization, and so must also belong to the generated set.

3. The set of subprocesses of the supremal element of the generated set can include processes that do not satisfy the characterization. For instance, all subprocesses of an accessible process are not necessarily accessible.

4. The supremal element is the largest subprocess of the generating element satisfying the characterization. Therefore, the two generated sets must be equal.

5. $\mathcal{X}(P)$ is the set of subprocesses of $P$ satisfying the characterization $\mathcal{X}(\cdot)$. Thus, when intersecting with $\mathcal{S}(\sup \mathcal{X}(P'))$, only those subprocesses of $\sup \mathcal{X}(P')$ satisfying the characterization remain. Of course, this is the set $\mathcal{X}(P')$, and it is an upper semilattice of $\mathcal{S}(P)$.

6. $\mathcal{X}(P')$ is the set of all subprocesses of $P$ satisfying the characterization $\mathcal{X}(\cdot)$, such that they are also subprocesses of $P'$. Note the equivalence between these last two expressions. Indeed $\mathcal{X}(P) \bigcap \mathcal{S}(P') = \mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{X}(P'))$.

Note that the characterization "subprocess" is union closed, as well as intersection closed. Thus, it is valid to substitute $\mathcal{S}(\cdot)$ for any of the characterizations above. For example, Lemma 4.5.1 shows that $\mathcal{X}(P') \subseteq \mathcal{S}(P'') \Rightarrow \sup \mathcal{X}(P') \leq \sup \mathcal{S}(P'') = P''$, which is used to prove Lemma 4.8 □

For the sake of brevity, in the following $P'$ and $P''$ denote arbitrary elements of $\mathcal{S}(P)$. When we say that " $\mathcal{X}(P)$ is a (characterized) upper semilattice", we implicitly also mean "under the subprocess relation". The sets $\mathcal{X}(P')$ and $\mathcal{Y}(P'')$ denote characterized upper semilattices with unique supremal elements $\sup \mathcal{X}(P')$ and $\sup \mathcal{Y}(P'')$, respectively. Note that it is always the case that $\sup \mathcal{X}(P') \leq P$.

**Lemma 4.6** The intersection of the upper semilattices $\mathcal{X}(P')$ and $\mathcal{Y}(P'')$ is also an upper semilattice.

**Proof.** We prove this by showing that the set $\mathcal{X}(P') \bigcap \mathcal{Y}(P'')$ is closed under union, that is for all elements $S', S'' \in \mathcal{X}(P') \bigcap \mathcal{Y}(P'')$ their union $S' \bigcup S'' \in \mathcal{X}(P') \bigcap \mathcal{Y}(P'')$.

When $S', S'' \in \mathcal{X}(P') \bigcap \mathcal{Y}(P'') \Leftrightarrow S', S'' \in \mathcal{X}(P') \wedge S', S'' \in \mathcal{Y}(P'')$. When $\mathcal{X}(P')$ and $\mathcal{Y}(P'')$ are closed under union, we have that $S' \bigcup S'' \in \mathcal{X}(P') \wedge S' \bigcup S'' \in \mathcal{Y}(P'') \Leftrightarrow S' \bigcup S'' \in \mathcal{X}(P') \bigcap \mathcal{Y}(P'')$. ■

**Remark.** Note that we use the same notation for subprocess union, as $S' \bigcup S''$ above, and (ordinary) set union, as $\mathcal{X}(P') \bigcup \mathcal{Y}(P'')$. We also use the same notation for subprocess intersection, like $S' \bigcap S''$, and (ordinary) set intersection, like $\mathcal{X}(P') \bigcap \mathcal{Y}(P'')$ above. The meaning will be clear form the context. □

Lemma 4.6 means that the supremal element $\sup [\mathcal{X}(P') \bigcap \mathcal{Y}(P'')]$ exists and is unique. Note that we will assume that any characterization is always satisfied by the null process.

That is, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ both include $\emptyset_P$, so that their (set) intersection $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$ is never empty. This significantly simplifies the proofs, while adding little loss of generalization. The characterizations pertaining to the supervisory control problem concern the states of the processes. Since the null process has no states, it satisfies all such characterizations. The null process as a solution to the problem of finding the supremal element of the intersection of two upper semilattices is deemed useless, however, and signals the fact that no (useful) solution exists.

Next we make the following observation concerning the relation between the sets $\mathcal{X}(P')$ and $\mathcal{Y}(P'')$ and their supremal elements.

**Lemma 4.7** For $\mathcal{X}(P')$ and $\mathcal{Y}(P'')$, the supremal element of $\mathcal{X}(P')$ is a subprocess of the supremal element of $\sup \mathcal{Y}(P'')$, if and only if $\mathcal{X}(P')$ is a subset of the set of subprocesses of $\mathcal{Y}(P'')$. That is,

$$\sup \mathcal{X}(P') \leq \sup \mathcal{Y}(P'') \Leftrightarrow \mathcal{X}(P') \subseteq \mathcal{S}(\sup \mathcal{Y}(P'')). \tag{4.6}$$

**Proof.** ($\Rightarrow$) It is obvious that $\sup \mathcal{X}(P') \leq \sup \mathcal{Y}(P'') \Leftrightarrow \forall P''' \in \mathcal{X}(P') \;\; P''' \leq \sup \mathcal{Y}(P'')$. By definition, $P''' \leq \sup \mathcal{Y}(P'') \Rightarrow P''' \in \mathcal{S}(\sup \mathcal{Y}(P''))$. Since this holds for all elements of $\mathcal{X}(P')$, it must hold that $\mathcal{X}(P') \subseteq \mathcal{S}(\sup \mathcal{Y}(P''))$.

($\Leftarrow$) Obviously, $\sup \mathcal{S}(\sup \mathcal{Y}(P'')) = \sup \mathcal{Y}(P'')$. Therefore, $\mathcal{X}(P') \subseteq \mathcal{S}(\sup \mathcal{Y}(P'')) \Rightarrow \sup \mathcal{X}(P') \leq \sup \mathcal{S}(\sup \mathcal{Y}(P'')) = \sup \mathcal{Y}(P'')$. ■

**Remark.** Note that $\sup \mathcal{X}(P') \leq \sup \mathcal{Y}(P'')$ does *not* imply that $\mathcal{X}(P')$ is a subset of $\mathcal{Y}(P'')$. There may very well be elements of $\mathcal{X}(P')$ that are not included in $\mathcal{Y}(P'')$. This is so, since not all subprocesses of $\sup \mathcal{Y}(P'')$ necessarily satisfy the characterization $\mathcal{Y}(\cdot)$. However, it always holds that $\mathcal{X}(P') \subseteq \mathcal{Y}(P'') \Rightarrow \sup \mathcal{X}(P') \leq \sup \mathcal{Y}(P'')$. □

Given a process $P$ and two union closed characterizations, $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$, we are looking for the largest subprocess of $P$ that satisfies both characterizations. This is of course the supremal element $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)]$. By Lemma 4.6 we know that this supremal element exists and is unique, since $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$ is also an upper semilattice. We also know that when the characterization is union closed, the set generated by an arbitrary process $P'$ is an upper semilattice. Specifically, this holds when $P' \leq P$, as is the case when $P' = \sup \mathcal{Y}(P)$, for instance. Then also, $\sup \mathcal{X}(P') \leq \sup \mathcal{X}(P)$. We can thus show the following subprocess ordering. Again, note that in $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$, $\bigcap$ denotes set intersection, while in $\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$ it denotes subprocess intersection.

**Lemma 4.8** For upper semilattices $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ with $P$ as generating element, the following ordering holds

$$\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{X}(\sup \mathcal{Y}(P)) \leq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P). \tag{4.7}$$

**Proof.** By Lemma 4.5.3 we have that $\mathcal{Y}(P) \subseteq \mathcal{S}(\sup \mathcal{Y}(P))$ so that $\mathcal{X}(P) \bigcap \mathcal{Y}(P) \subseteq \mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{Y}(P))$. From Lemma 4.5.5 we get that $\mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{Y}(P)) = \mathcal{X}(\sup \mathcal{Y}(P))$, and therefore $\mathcal{X}(P) \bigcap \mathcal{Y}(P) \subseteq \mathcal{X}(\sup \mathcal{Y}(P))$. By Lemma 4.5.1 we finally have that the left subprocess relation $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{X}(\sup \mathcal{Y}(P))$ holds.

Obviously $\sup \mathcal{X}(\sup \mathcal{Y}(P)) \leq \sup \mathcal{X}(P)$. From Lemma 4.5.1 with $P' = P'' = P$ and $\mathcal{Y}(\cdot) = \mathcal{S}(\cdot)$, we can derive that $\sup \mathcal{X}(\sup \mathcal{Y}(P)) \leq \sup \mathcal{Y}(P)$. From Theorem 2.52.3 we then know that $\sup \mathcal{X}(\sup \mathcal{Y}(P)) \leq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$.

■

**Remark.** It should be clear that this ordering also holds if we exchange the order of calculation of the supremal elements of $\sup \mathcal{X}(\sup \mathcal{Y}(P))$. That is,

$$\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{Y}(\sup \mathcal{X}(P)) \leq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P), \tag{4.8}$$

also holds, as is easily verified.

□

Now we have an ordering of a number of supremal elements that can all be calculated, either by brute force, calculating all elements of both sets, or by more effective means, such as calculating the supremal elements of both sets and intersecting these, or calculating the supremal element of the upper semilattice obtained from using the supremal element of the other set as generating element. All three of these can be calculated, but to efficiently find the supremal element of both sets we must use as little computing power as possible. Since the number of subprocesses of either set may be very large, the brute force method is really no alternative. Even if an efficient way of calculating the supremal elements of either set, without calculating the whole set, is given, calculating the supremal elements of both sets and intersecting these, requires the operation of intersection. Calculating $\sup \mathcal{X}(\sup \mathcal{Y}(P))$ does not include this intersection of two processes, and so this would seem to be the most effective way to calculate the sought supremal element, *if this was our answer*.

Fortunately, when $\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P)$ it is.

**Theorem 4.9** For two union closed characterizations $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$, the supremal element of the intersection $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$ of the upper semilattices $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ with $P$ as generating element, is equal to the supremal element of one of the characterized upper semilattices with the supremal element of the other set as generating element, if the intersection of the supremal elements of the respective sets is an element of the intersection $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$. That is,

$$\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P) \Rightarrow \sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{X}(\sup \mathcal{Y}(P)). \tag{4.9}$$

**Proof.** By Lemma 4.8 we know that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$. Thus, when $\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P)$ we must have that $\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$ is equal to $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)]$. Obviously, when this holds, the subprocess ordering of Lemma 4.8 collapses, so that all the supremal subprocesses are equal, and hence

$$\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{X}(\sup \mathcal{Y}(P)) = \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P). \tag{4.10}$$

■

**Remark.** This means that if we can show that the intersection of the supremal elements of both sets is an element of both sets, then we can calculate the supremal element of

the intersection of the sets by first calculating the supremal element of one set, and then calculate the supremal subprocess of this element satisfying the other characterization. Calculating the supremal element of a subprocess $P' \leq P$ satisfying a characterization, would not be any different from calculating the supremal element of $P$ satisfying that characterization. On the contrary it would in general require less number of operations.

Since Lemma 4.8 hold equally for $\sup \mathcal{Y}(\sup \mathcal{X}(P))$, it does not matter in which order the supremal elements are calculated. That is, Theorem 4.9 also holds for $\sup \mathcal{Y}(\sup \mathcal{X}(P))$. However, note that this is so only when $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$. This is an important fact which will be generalized to the case when $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \neq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$. □

In the general case it does not necessarily hold that $\sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P)$. Therefore, we are not guaranteed to acquire the sought supremal element by calculating $\sup \mathcal{X}(\sup \mathcal{Y}(P))$. We will show, though, that iteratively repeating the steps of calculating the supremal elements of the previous calculation will eventually give us the supremal element of the intersection of the sets. For this, we first define the following operator, applying the calculation of the supremal elements *interweavingly*, that is, first $\sup \mathcal{Y}(P)$, then $\sup \mathcal{X}(\sup \mathcal{Y}(P))$, then $\sup \mathcal{Y}(\cdot)$ of this element, etc.

**Definition 4.10 The Supremal Operator**

For two union closed characterizations $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$, define on the upper semilattices, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$, with generating element $P$, the *supremal operator*, $\sup \mathcal{XY}_n(\cdot)$, as

$$\begin{aligned} \sup \mathcal{XY}_0(P) &= \sup \mathcal{Y}(P) \\ \sup \mathcal{XY}_1(P) &= \sup \mathcal{X}(\sup \mathcal{Y}(P)) \\ \sup \mathcal{XY}_{n+1}(P) &= \begin{cases} \sup \mathcal{X}(\sup \mathcal{XY}_n(P)) & n \text{ even} \\ \sup \mathcal{Y}(\sup \mathcal{XY}_n(P)) & n \text{ odd} \end{cases} \end{aligned} \tag{4.11}$$

The index $n$ will be called the *iteration index*.

**Remark.** It is obvious that $\sup \mathcal{XY}_{n+1}(P) \leq \sup \mathcal{XY}_n(P) \leq \sup \mathcal{XY}_{n-1}(P)$, since each increase in iteration index in essence adds one constraint to the generated set.

Note that when the iteration index, $n$, is odd, the outermost calculation of $\sup \mathcal{XY}_n(P)$ calculates $\sup \mathcal{X}(\cdot)$, and when $n$ is even it calculates $\sup \mathcal{Y}(\cdot)$. Thus, the following are immediate consequences of the definition.

$$\begin{aligned} n \text{ odd}: \quad & \sup \mathcal{X}(\sup \mathcal{XY}_n(P)) = \sup \mathcal{XY}_n(P) \\ & \sup \mathcal{Y}(\sup \mathcal{XY}_{n+1}(P)) = \sup \mathcal{XY}_{n+1}(P) \end{aligned} \tag{4.12}$$

We could also have defined a complementary supremal operator, $\sup \mathcal{YX}_n(\cdot)$, which started by calculating $\sup \mathcal{X}(\cdot)$ of the generating element. In fact, all the properties that are proved for $\sup \mathcal{XY}_n(\cdot)$ hold with duality for $\sup \mathcal{YX}_n(\cdot)$. Note though that, depending on the ordering and the characterizations, it may in some cases be more efficient to use one of the operators instead of the other. See for instance Example 4.2. When and why this is so has not been investigated in any depth. One case where it is known which one is more effective, is when one of the sets contains the generating element as the supremal element. Then it is more effective to calculate the supremal element of the other set first.

Note also that $\sup \mathcal{YX}_n(\cdot)$ is not necessarily equal to $\sup \mathcal{XY}_n(\cdot)$, on the contrary. This is also shown in Example 4.2. □

Next we show that when applying the iterative supremal operator to two upper semilattices, if and once a supremal element is equal to the next calculated supremal element, then all other supremal elements will be equal to this element. That is, we start out with a process $P$ as generating element and calculate the supremal element with respect to one characterization. Then we alternatively calculate the supremal element of the subset obtained from using the supremal element of the other characterized set as generating element. If an element is calculated that remains fixed, then it is fixed no matter how many times more we perform the iteration.

**Lemma 4.11** For two upper semilattices, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$, we have that

$$\sup \mathcal{XY}_{n+1}(P) = \sup \mathcal{XY}_n(P) \Leftrightarrow \sup \mathcal{XY}_{n+k}(P) = \sup \mathcal{XY}_n(P) \quad \forall k \geq 1. \tag{4.13}$$

**Proof.** We will show only for odd $n$. The result for the case when $n$ is even follows as a dual. Just exchange " sup 1" for " sup 2", and vice versa.

($\Rightarrow$) When $n$ is odd, we have by (4.12) that $\sup \mathcal{X}(\sup \mathcal{XY}_n(P)) = \sup \mathcal{XY}_n(P)$ and that $\sup \mathcal{Y}(\sup \mathcal{XY}_{n+1}(P)) = \sup \mathcal{XY}_{n+1}(P)$. When $\sup \mathcal{XY}_n(P) = \sup \mathcal{XY}_{n+1}(P)$ we have that

$$\begin{aligned} \sup \mathcal{XY}_{n+1}(P) &= \sup \mathcal{Y}(\sup \mathcal{XY}_{n+1}(P)) \\ &= \sup \mathcal{Y}(\sup \mathcal{XY}_n(P)) \\ &= \sup \mathcal{XY}_n(P) \end{aligned} \tag{4.14}$$

so that $\sup \mathcal{XY}_n(P)$ is the supremal element of both the characterized subsets $\mathcal{X}(\sup \mathcal{XY}_n(P))$ and $\mathcal{Y}(\sup \mathcal{XY}_n(P))$. Therefore, no matter how many times we calculate the supremal element of the sets obtained from $\sup \mathcal{XY}_n(P)$ as generating element, $\sup \mathcal{XY}_n(P)$ will always be the result. This then, is our fixpoint.

($\Leftarrow$) When $\forall k \geq 1$ $\sup \mathcal{XY}_n(P) = \sup \mathcal{XY}_{n+k}(P)$ this specifically holds for $k = 1$, so that $\sup \mathcal{XY}_n(P) = \sup \mathcal{XY}_{n+1}$. ■

From Lemma 4.11 we also have the following immediate corollary.

**Corollary 4.12** For two upper semilattices, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$, we have that

$$\begin{aligned} \sup \mathcal{XY}_{n+1}(P) = \sup \mathcal{XY}_n(P) \Leftrightarrow \\ \sup \mathcal{XY}_n(P) = \sup \mathcal{X}(\sup \mathcal{XY}_n(P)) = \sup \mathcal{Y}(\sup \mathcal{XY}_n(P)) \end{aligned} \tag{4.15}$$

Now we know that if the iterative supremal operator $\sup \mathcal{XY}_n(P)$ finds an element that is equal from one iteration to the next, then this element is such that all consequent iterations will return that element. Now we have to show that there does indeed exist such an element and it will be found by the supremal operator , so that we can guarantee that the supremal operator terminates in a finite number of iterations. We prove this by showing that there does always exist a lower bound for the supremal operator, such that, for any $n$, this lower bound is a subset of $\sup \mathcal{XY}_n(P)$. It so happens that this lower bound is in fact the element that we are looking for, the supremal element of the intersection of both characterized sets.

**Example 4.2 Calculating Supremal Element**

This example shows an application of using the supremal operator to calculate the supremal element of the intersection of two upper semilattices, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$.

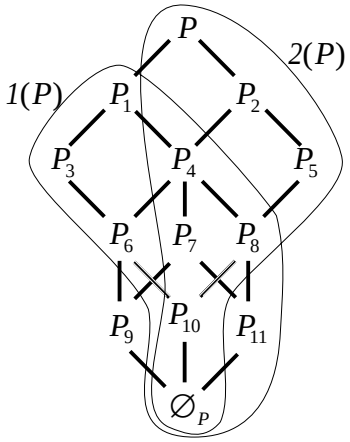



**Figure 4.2:** The sets $\mathcal{S}(P)$, $\mathcal{X}(P)$ and $\mathcal{Y}(P)$. $P_1 = \sup \mathcal{X}(P)$ and $P = \sup \mathcal{Y}(P)$. Note that $P_6$ and $P_7$ both belong to $\mathcal{X}(P)$, but $P_6 \bigcap P_7 = P_9 \notin \mathcal{X}(P)$, thus $\mathcal{X}(P)$ is not closed under subprocess intersection. Neither is $\mathcal{Y}(P)$, since $P_{11} \notin \mathcal{Y}(P)$. However, it is easily verified that both sets are closed under subprocess union.

We have the generated set of $\mathcal{X}(P) = \{P_1, P_3, P_4, P_6, P_7, P_8, P_{10}, P_{11}, \emptyset_P\}$ and for $\mathcal{Y}(P) = \{P, P_2, P_4, P_5, P_7, P_8, P_{10}, \emptyset_P\}$, and, of course, the supremal element of their intersection is $\sup[\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup\{P_4, P_7, P_8, P_{10}, \emptyset_P\} = P_4$. The supremal element of $\mathcal{X}(P) \bigcap \mathcal{Y}(P)$ is obtained by iteratively calculating supremal elements as follows.

1. $\sup \mathcal{XY}_1(P) = \sup \mathcal{X}(\sup \mathcal{Y}(P)) = \sup \mathcal{X}(P) = P_1$, since $\sup \mathcal{Y}(P) = P$.
2. $\sup \mathcal{XY}_2(P) = \sup \mathcal{Y}(\sup \mathcal{XY}_1(P)) = \sup[\mathcal{Y}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_1(P))] =$
   $\sup\{P_4, P_7, P_8, P_{10}, \emptyset_P\} = P_4$
3. $\sup \mathcal{XY}_3(P) = \sup \mathcal{X}(\sup \mathcal{XY}_2(P)) = \sup[\mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_2(P))] =$
   $\sup\{P_4, P_6, P_7, P_8, P_{10}, P_{11}, \emptyset_P\} = P_4$

We can note that the sets $\mathcal{XY}_2(P)$ and $\mathcal{XY}_3(P)$ are not equal, even though their supremal elements are. However, for all even $n \geq 2$, $\mathcal{XY}_2(P) = \mathcal{XY}_n(P)$, and for all odd $n \geq 3$, $\mathcal{XY}_3(P) = \mathcal{XY}_n(P)$. Furthermore, as shown by Lemma 4.13, $\sup[\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_n(P)$ for any $n$.

Note also that using the complementary supremal operator $\sup \mathcal{YX}_n(P)$, which begins by calculating $\sup \mathcal{X}(P) = P_1$, convergence would have been reached in just three supremal calculations, instead of the four required above. This is so since $\sup \mathcal{YX}_1(P) = \sup \mathcal{Y}(P_1) = P_4$, and then $\sup \mathcal{YX}_2(P) = \sup \mathcal{X}(P_4) = P_4$, while $\sup \mathcal{Y}(P) = P$ and $\sup \mathcal{XY}_1(P) = \sup \mathcal{X}(P) = P_1$. Thus, $\sup \mathcal{Y}(P)$ conveys no new information, and is in practice a redundant calculation.

■

**Lemma 4.13** For upper semilattices $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ with $P$ as generating element, and a supremal operator $\sup \mathcal{XY}_n(P)$, the following ordering holds for any iteration index $n$

$$\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_n(P) \tag{4.16}$$

**Proof.** For $n = 0$ we have that $\sup \mathcal{XY}_0(P) = \sup \mathcal{Y}(P)$. Obviously, $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_0(P)$.

For $n = 1$, that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_1(P) = \sup \mathcal{X}(\sup \mathcal{Y}(P))$ was shown in Lemma 4.8.

The rest will follow by induction from these two properties.

Assume that Lemma 4.8 holds for $n = m$, that is $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_m(P)$. By Lemma 4.7 this is equivalent to $\mathcal{X}(P) \bigcap \mathcal{Y}(P) \subseteq \mathcal{S}(\sup \mathcal{XY}_m(P))$. Since it always holds that $\mathcal{X}(P) \bigcap \mathcal{Y}(P) \subseteq \mathcal{X}(P)$, we have that $\mathcal{X}(P) \bigcap \mathcal{Y}(P) \subseteq \mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_m(P))$. From Lemma 4.5.1, this implies that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup [\mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_m(P))]$.

For $n = m+1$ we have that (when $m$ is even)[1] $\sup \mathcal{XY}_{m+1}(P) = \sup \mathcal{X}(\sup \mathcal{XY}_m(P))$. From Lemma 4.5.6 $\sup \mathcal{X}(\sup \mathcal{XY}_m(P)) = \sup [\mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_m(P))]$. By the assumption above we have that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_m(P)$ implies that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup [\mathcal{X}(P) \bigcap \mathcal{S}(\sup \mathcal{XY}_m(P))] = \sup \mathcal{XY}_{m+1}(P)$.

By the principle of induction, the results above prove that Lemma 4.13 holds. ∎

**Remark.** Of course, we will always have that $\sup \mathcal{XY}_n(P) \leq \sup \mathcal{X}(P) \bigcap \sup \mathcal{Y}(P)$, see Theorem 2.52.3, so that the ordering of Lemma 4.13 is a generalization of Lemma 4.8. This lemma only concerns the special case of $n = 1$.

Note also that Lemma 4.13, together with the remark to Definition 4.10, means that for any $n$

$$\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_{n+1}(P) \leq \sup \mathcal{XY}_n(P) \leq \sup \mathcal{XY}_0(P) \tag{4.17}$$

□

We know now that when the characterizations $\mathcal{X}(\cdot)$ and $\mathcal{Y}(\cdot)$ are union closed and such that they are both satisfied by the null process, there does exist a unique supremal element that is contained in both sets $\mathcal{X}(P)$ and $\mathcal{Y}(P)$. Now we want to find this element by some effective calculation.

Intuitively, since the lower bound of the supremal operator is the element that is sought, this element is equal to the fixpoint of the supremal operator. If this were the case, then we would indeed have an effective way of calculating this element, only calculating as many elements of the sets $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ as is absolutely necessary to find the supremal element contained in both sets. It can be shown that this is so.

**Theorem 4.14** For upper semilattices $\mathcal{X}(P)$ and $\mathcal{Y}(P)$ with $P$ as generating element, and the supremal operator $\sup \mathcal{XY}_n(P)$, the supremal element of the intersection of the sets is equal to the fixpoint calculated by the supremal operator. That is,

$$\sup \mathcal{XY}_{n+1}(P) = \sup \mathcal{XY}_n(P) \Leftrightarrow \sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{XY}_n(P) \tag{4.18}$$

[1] Again, it is straightforward to show that this also holds for odd $m$.

**Proof.** Again, we show only for even iteration index. The result follows similarly for odd $n$.

($\Rightarrow$) When $\sup \mathcal{XY}_n(P) = \sup \mathcal{XY}_{n+1}(P)$ we know by Corollary 4.12 that

$$\sup \mathcal{XY}_n(P) = \sup \mathcal{X}(\sup \mathcal{XY}_n(P)) = \sup \mathcal{Y}(\sup \mathcal{XY}_n(P)), \tag{4.19}$$

and thus $\sup \mathcal{XY}_n(P) \in \mathcal{X}(P)$, as well as $\sup \mathcal{XY}_n(P) \in \mathcal{Y}(P)$. Therefore, $\sup \mathcal{XY}_n(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P)$. Since we by Lemma 4.13 know that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_n(P)$, for any $n$, when $\sup \mathcal{XY}_n(P) \in \mathcal{X}(P) \bigcap \mathcal{Y}(P)$ we must have that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{XY}_n(P)$.

($\Leftarrow$) By (4.17) we know that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] \leq \sup \mathcal{XY}_{n+1}(P) \leq \sup \mathcal{XY}_n(P)$. Thus, when $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{XY}_n(P)$ it must also hold that $\sup [\mathcal{X}(P) \bigcap \mathcal{Y}(P)] = \sup \mathcal{XY}_{n+1}(P)$, and thus $\sup \mathcal{XY}_{n+1}(P) = \sup \mathcal{XY}_n(P)$. ■

Theorem 4.14 tells us that the iteration of the supremal operator will always find a fixpoint for a finite and bounded iteration index, and when it does, this fixpoint is the supremal element of the intersection of the characterized upper semilattices $\mathcal{X}(P)$ and $\mathcal{Y}(P)$.

In this section we have shown that for arbitrary upper semilattices with common universe of discourse, the supremal element of the intersection of any two such upper semilattices can be efficiently calculated by an iterative operator. This is a general result that has applications to many areas, not only that of subprocess characterizations.

In the next sections, we will use the algorithm defined by this theorem, to calculate the supremal trim and complete subprocess of a given process that is to act as a supervisor for a plant. Furthermore, Theorem 4.9 will be used in calculating the supremal trim subprocess, since it can be shown that the sets of accessible and coaccessible subprocesses hold the required property that the intersection of the supremal elements is both accessible and coaccessible.

## 4.2 The Supremal Trim Subprocess

As we are set out to show that we can calculate the supremal trim subprocess of a given process $S$, we begin by defining the set of all trim subprocesses of $S$. Since trimness of a transition machine is equivalent to accessibility and coaccesibility we will begin by defining these sets.

**Definition 4.15 Sets of All Accessible, Coaccessible and Trim Subprocesses**

For a process $S$ set of *all accessible subprocesses* is defined as

$$\mathcal{A}(S) = \{\, S' \in \mathcal{S}(S) |\, S' \text{ is accessible}\}, \tag{4.20}$$

the set of *all coaccessible subprocesses* is defined as

$$\bar{\mathcal{A}}(S) = \{\, S' \in \mathcal{S}(S) |\, S' \text{ is coaccessible}\}, \tag{4.21}$$

and the set of *all trim subprocesses* is defined as

$$\mathcal{T}(S) = \{\, S' \in \mathcal{S}(S) |\, S' \text{ is trim}\}, \tag{4.22}$$

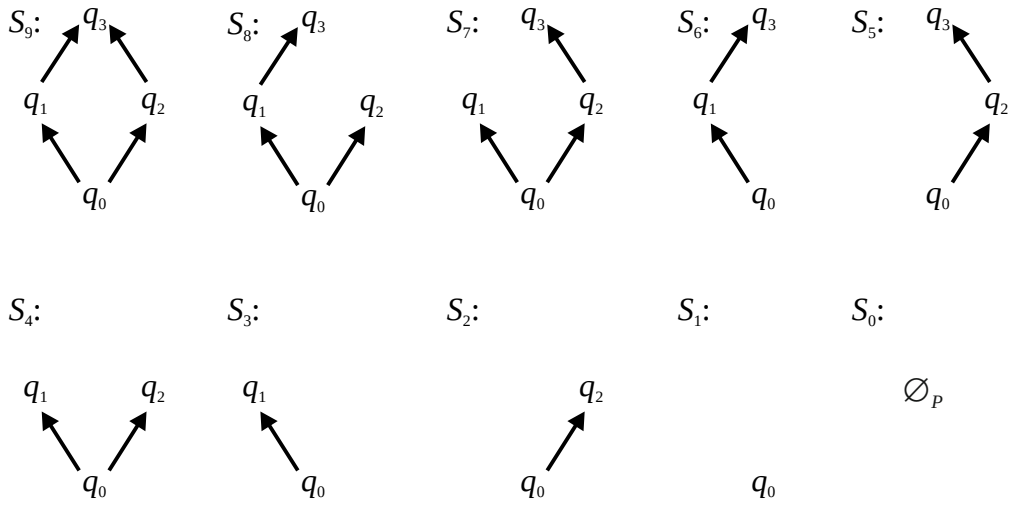



**Figure 4.3:** The set $\mathcal{A}(P)$ of all accessible subprocesses of $P = S_9$. Note that, for instance, $S_5 \bigcap S_8$ is not accessible since $q_3 \in Q_{S_5} \bigcap Q_{S_8}$ but no edge $e'$ with $q_3 = last(e')$ exists in $E_{S_5} \bigcap E_{S_8}$. Note though, that the union of any two elements of $\mathcal{A}(P)$ is still in $\mathcal{A}(P)$.

**Remark.** We use the notation $\bar{\mathcal{A}}(\cdot)$ since coaccessibility can in some sense be regarded as the complement to accessibility. It is an obvious fact that the set $\mathcal{A}(S)$ is closed under union. This follows from Lemma 2.53. When a subprocess is accessible, every state is reachable by some trace. These traces do not disappear under union, so in the union all states are also accessible. See also Figures 4.3 and 4.4. Closedness of $\bar{\mathcal{A}}(S)$ under union again follows from the fact that all traces survive under union, see Lemma 2.53. By Definition 2.18 we know that the set of trim subprocesses is the intersection of the set of accessible and coaccessible subprocesses, $\mathcal{T}(S) = \mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S)$. By Lemma 4.6, and the above comments, we know that $\mathcal{T}(S)$ is also closed under subprocess union. □

By the remark above we know that $\mathcal{A}(S)$, $\bar{\mathcal{A}}(S)$ and $\mathcal{T}(S)$ are upper semilattices under the operation of union. Furthermore, the null process is both accessible and coaccessible, so that it belongs to both sets. Therefore, we can use the lemmas and theorems derived in Section 4.1.

Since $\mathcal{T}(S) = \mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S)$ we could find $\sup \mathcal{T}(S)$ by generating the sets of accessible and coaccessible subprocesses and take the supremal element of their intersection. However, we want to do this more efficiently, not having to generate the entire sets of $\mathcal{A}(S)$ and $\bar{\mathcal{A}}(S)$. It seems that Definition 2.18 strongly suggests that the supremal trim subprocess of a given process, can be obtained by intersecting the supremal accessible and the supremal coaccessible subprocesses. As it turns out, this is so. But first, let us give algorithms for calculating the supremal accessible and the supremal coaccessible subprocesses.

The following algorithm is a variant of an algorithm given by Eilenberg (1974), though without proof.

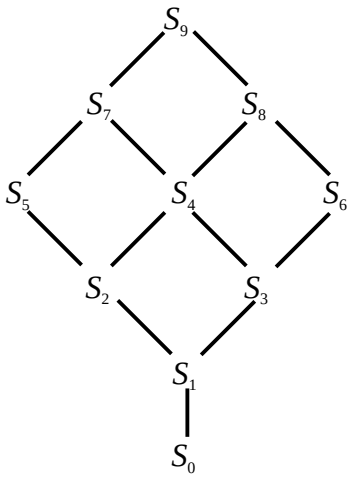



**Figure 4.4:** The ordering of the subprocesses in $\mathcal{A}(P)$.

**Algorithm 4.16 Supremal Accessible Subprocess**

Define an operator $\Theta(Q, Q_i) : 2^{Q_S} \times 2^{Q_S} \to 2^{Q_S}$ on a state-space $Q_S$ of a transition machine $S$ such that

$$\Theta(Q, Q_i) = \{q \in (Q_S - Q) \,|\exists (q', \sigma, q) \in E_S \text{ such that } q' \in Q_i\} \tag{4.23}$$

1. Set $Q_0 \leftarrow I_S$
2. Iterate $Q_{i+1} \leftarrow \Theta(\bigcup_{j=0}^{i} Q_j, Q_i)$ until $Q_{i+1} = \emptyset$
3. $Q_{ac} = \bigcup_{j=0}^{i} Q_j$ and then $E_{ac} = \{e \in E_S \,|first(e) \in Q_{ac} \wedge last(e) \in Q_{ac}\}$

The generated accessible subprocess $S_{ac} \leq S$ then has the state-space $Q_{ac}$ and the edgeset $E_{ac}$.

**Remark.** The $\Theta$ operator picks out those states of $Q_S$ not in $Q$, such that they are reachable from the states of $Q_i$. Beginning with the initial states, $\Theta$ is used iteratively to return sets of states reachable in one "step", by some edge, from the states of the former iteration, until no more states can be reached from the present state subspace. The operator is careful not to pick out states that are already known to be accessible. Thus, even if there existed an edge from some state of the present state-space into some former state-space, that reached state is not picked out again. The union over these state subspaces of $Q_S$ are the accessible states of $Q_S$. Removing all other states, which are not reachable from the initial states, and all edges concerning these states generates the accessible subprocess $S_{ac} \leq S$. □

**Lemma 4.17** Algorithm 4.25 terminates in *at most* $|Q_S|$ iterations.

**Proof.** All of the $Q_i$ are disjoint, since $q \in Q_{i+1} \Leftrightarrow q \in Q_S \wedge q \notin Q_0 \bigcup \ldots \bigcup Q_i$. Therefore (for $n = |Q_S|$) $Q_n = Q_S - \bigcup_{j=0}^{n-1} Q_j = \emptyset$. ∎

**Remark.** This means that *if* the iteration goes on until $i = |Q_S| - 1$, *then* the algorithm necessarily terminates. Note though, that termination after $|Q_S|$ iterations is a worst case scenario with all $Q_i$ singleton. Normally the algorithm will stop before the entire state-space is exhausted, since once $Q_i = \emptyset$ $Q_{i+k} = \emptyset$ for $k = 1, 2, \ldots$ □

Now we must also show that the accessible subprocess $S_{ac} \leq S$ found by Algorithm 4.16 is indeed the *supremal* accessible subprocess. That is, we have to show that $S_{ac} \leq \sup \mathcal{A}(S)$ and $\sup \mathcal{A}(S) \leq S_{ac}$.

**Lemma 4.18** Algorithm 4.16 finds the supremal accessible subprocess of $S$

**Proof.** It is obvious that the states of $Q_{ac}$ are accessible, so that $S_{ac}$ is an accessible subprocess of $S$, and hence $S_{ac} \leq \sup \mathcal{A}(S)$. Since both $\sup \mathcal{A}(S)$ and $S_{ac}$ are accessible, we know that their tracesets hold all relevant information. Thus, $tr(S_{ac}) \subseteq tr(\sup \mathcal{A}(S))$, and we have to show that $tr(\sup \mathcal{A}(S)) \subseteq tr(S_{ac})$.

Assume that $tr(\sup \mathcal{A}(S)) \not\subseteq tr(S_{ac})$. Then there exists a trace $t \in tr(\sup \mathcal{A}(S))$ but $t \notin tr(S_{ac})$. This trace we can writes as a sequence of well-ordered edges, $t = e_0 e_1 \ldots e_m$, where $e_i \in E_S$ for $i = 0, 1, \ldots, m$. Obviously, $first(e_0) \in Q_0$ and $Q_0 \subseteq Q_{\sup \mathcal{A}(S)}$ as well as $Q_0 \subseteq Q_{S_{ac}}$. On the first iteration, we calculate $Q_1 = \Theta(Q_0, Q_0)$. That is

$$Q_1 = \{q \in Q_S - Q_0 \,|\exists (q', \sigma, q) \in E_S \;\; q' \in Q_0\}. \tag{4.24}$$

Clearly, since $first(e_0) \in Q_0$, $first(e_1) \in Q_1$ and $e_0 e_1 \in tr(S_{ac})$. Continuing this iteration we get that $t \in tr(S_{ac})$. This contradiction shows that $S_{ac}$ is indeed equal to $\sup \mathcal{A}(S)$. ∎

To find the supremal coaccessible subprocess of $S$, we essentially do the same thing as in Algorithm 4.16, only backwards. We start with the marked states and iterate until no more states can be found that reach the states picked out by the former iteration.

**Algorithm 4.19 Supremal Coaccessible Subprocess**

Define an operator $\Omega(Q, Q_i) : 2^{Q_S} \times 2^{Q_S} \rightarrow 2^{Q_S}$ on a state-space $Q_S$ of a transition machine $S$ such that

$$\Omega(Q, Q_i) = \{q \in (Q_S - Q) \,|\exists (q, \sigma, q') \in E_S \;\; \text{such that } q' \in Q_i\} \tag{4.25}$$

1. Set $Q_0 \leftarrow M_S$
2. Iterate $Q_{i+1} \leftarrow \Omega(\bigcup_{j=0}^{i} Q_j, Q_i)$ until $Q_{i+1} = \emptyset$
3. $Q_{co} = \bigcup_{j=0}^{i} Q_j$ and then $E_{co} = \{e \in E_S \,|first(e) \in Q_{co} \wedge last(e) \in Q_{co}\}$

**Example 4.3 Supremal Trim Subprocess**

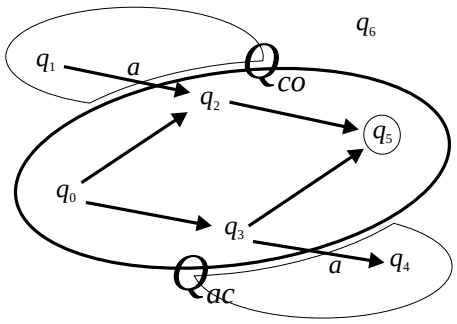



**Figure 4.5:** The accessible and coaccessible subprocesses of a transition machine $S$. The accessible states are $Q_{ac} = Q_S - \{q_1, q_6\}$ and the coaccessible states are $Q_{co} = Q_S - \{q_4, q_6\}$.

In Figure 4.5 is shown a transition machine $S$. Its initial state is $q_0$, and its only marked state is $q_5$. The states $\{q_0, q_2, q_3, q_5\} = Q_{ac} \bigcap Q_{co}$ are both accessible and coaccessible, while $q_1 \in Q_{co}$ and $q_4 \in Q_{ac}$. The state $q_6$ is, of course, neither accessible nor coaccessible.

Note that the subprocess intersection sup
$A(S) \bigcap \sup \bar{\mathcal{A}}$ yields a trim subprocess of $S$. However, the intersection between the accessible subprocess $A$ with $Q_A = \{q_0, q_2, q_3\}$ and the coaccessible subprocess $C$ with $Q_C = \{q_2, q_3, q_5\}$ does not result in a trim subprocess of $S$. $Q_A \bigcap Q_C = \{q_2, q_3\}$, so that $A \bigcap C$ can be neither accessible nor coaccessible.

■

The supremal coaccessible subprocess $S_{co} \leq S$ then has the state-space $Q_{co}$ and the edge-set $E_{co}$.

The proof for termination of Algorithm 4.19 and the proof of it finding the supremal coaccessible subprocess go along the same lines as shown for Algorithm 4.16. We will not go into this again.

Now we have effective ways of calculating the supremal accessible and the supremal coaccessible subprocess of a given process $S$. When using these algorithms to calculate the supremal trim element, that is, $\sup\left[\mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S)\right]$, we want to be as effective as possible. From Theorem 4.9 we know that if we can show that $\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)$ is both accessible and coaccessible, then the supremal trim element can be calculated as $\sup \mathcal{A}(\sup \bar{\mathcal{A}}(S))$ or $\sup \bar{\mathcal{A}}(\sup \mathcal{A}(S))$.

**Lemma 4.20** For a given transition machine $S$, the intersection of the supremal accessible and the supremal coaccessible subprocesses is both accessible and coaccessible. That is,

$$\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S) \in \mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S). \tag{4.26}$$

**Proof.** All states of $\sup \mathcal{A}(S)$ are accessible and $Q_{\sup \mathcal{A}(S)}$ is by definition the largest subset of accessible states of $Q_S$. Similarily, all states of $\sup \bar{\mathcal{A}}(S)$ are coaccessible and $Q_{\sup \bar{\mathcal{A}}(S)}$ is

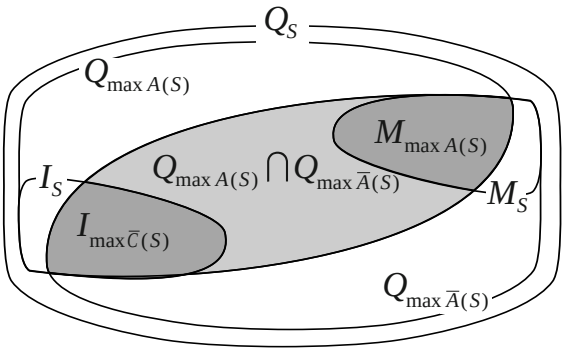



**Figure 4.6:** The relations between the initial and marked states of $\sup \mathcal{A}(S)$ and $\sup \bar{\mathcal{A}}(S)$. Obviously, $\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)$ is both accessible and coaccessible.

the largest subset of $Q_S$ that is coaccessible. Since all states of $I_S$ are accessible, $I_{\sup \mathcal{A}(S)} = I_S$ and $I_{\sup \bar{\mathcal{A}}(S)} \subseteq I_{\sup \mathcal{A}(S)}$, thus $I_{\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)} = I_{\sup \mathcal{A}(S)} \bigcap I_{\sup \bar{\mathcal{A}}(S)} = I_{\sup \bar{\mathcal{A}}(S)}$. Likewise, since all states of $M_S$ are coaccessible, $M_{\sup \bar{\mathcal{A}}(S)} = M_S$ and $M_{\sup \mathcal{A}(S)} \subseteq M_{\sup \bar{\mathcal{A}}(S)}$, and thus $M_{\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)} = M_{\sup \mathcal{A}(S)} \bigcap M_{\sup \bar{\mathcal{A}}(S)} = M_{\sup \mathcal{A}(S)}$. Note that this is not necessarily so for accessible and coaccessible subprocesses that are not supremal. See also Figure 4.6.

We know by Lemma 2.55 that $tr(\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)) = tr(\sup \mathcal{A}(S)) \bigcap tr(\sup \bar{\mathcal{A}}(S))$. Since $\sup \mathcal{A}(S)$ is the *supremal* accessible subprocess of $S$, no traceset of any subprocess of $S$ can ever reach outside $tr(\sup \mathcal{A}(S))$. Therefore, $tr(\sup \bar{\mathcal{A}}(S)) \subseteq tr(\sup \mathcal{A}(S))$, so that $tr(\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)) = tr(\sup \bar{\mathcal{A}}(S))$, meaning that the accessible states of the subprocess $\sup \mathcal{A}(S) \bigcap \sup C'(S)$ are exactly those that are accessible by $\sup \bar{\mathcal{A}}(S)$. Naturally, such a state $q$ is also coaccessible in $\sup \bar{\mathcal{A}}(S)$, so that there exists some trace $t$ of $E^*_{\sup \bar{\mathcal{A}}(S)} \subseteq E^*_S$ taking $q$ into $M_S$. Since $q$ is accessible, $t$ also exists in $E^*_{\sup \mathcal{A}(S)} \subseteq E^*_S$, and thus all reachable states of $Q_{\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)}$ are also coaccessible.

By sreasoning we can show that all coaccessible states of $Q_{\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)}$ are also accessible. Thus, $\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)$ is both accessible and coaccessible.

■

**Remark.** Therefore, $\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S)$ is trim and $\sup \mathcal{A}(S) \bigcap \sup \bar{\mathcal{A}}(S) \in \mathcal{T}(S)$. By Theorem 4.9 we then know that $\sup \mathcal{T}(S) = \sup \mathcal{A}(\sup \bar{\mathcal{A}}(S)) = \sup \bar{\mathcal{A}}(\sup \mathcal{A}(S))$.

Note that the supremality of $\sup \mathcal{A}(S)$ and $\sup C'(S)$ is of essence here. The intersection of *any* accessible subprocess with *any* coaccessible subprocess, does not necessarily yield a trim subprocess. On the contrary, see Example 4.3. □

Now we know that the supremal trim subprocess of a given process can be found by calculating the supremal accessible subprocess of the supremal coaccessible subprocess of $S$, or vice versa. Therefore, the algorithm for calculating the supremal trim subprocess is very simple.

**Algorithm 4.21 Supremal Trim Subprocess**

The following two steps calculate the supremal trim subprocess of a given process $S$.

1. Calculate $\sup \bar{\mathcal{A}}(S)$ using Algorithm 4.19.
2. Calculate $\sup \mathcal{A}(\sup \bar{\mathcal{A}}(S))$ using Algorithm 4.16.

**Proof.** The proof of this algorithm is given by Theorem 4.9 together with the fact that the intersection of the supremal accessible and the supremal coaccessible subprocess is trim, as shown by Lemma 4.20. Naturally, the proofs of Algorithm 4.19 and Algorithm 4.16 are also needed. ■

**Remark.** Of course, it does not matter if we calculate the supremal accessible or the supremal coaccessible subprocess of $S$ first. In Algorithm 4.21 we have just arbitrarily chosen to calculate the supremal coaccessible subprocess first. However, there may be performance gain in choosing to calculate one before the other. This has not been investigated, though. □

## 4.3 The Complete Accessible Subprocess

In this section we will give an algorithm for calculating the supremal complete subprocess of a given process $S$. Complete, that is, with respect to some plant $P$ with uncontrollable events. In this section we will make the following three assumptions.

- We will assume that $P$ is deterministic, though $S$ may be non-deterministic.
- It will be assumed that $S$ refines $P$, and thus, $L(S) \subseteq L(P)$.
- $P$ and the subprocesses of $S$ will all be assumed to be accessible.

The loss of generality induced by these assumptions is of minor practical consequence. The main loss of generality comes from the assumption of the plant to be deterministic. We believe that the algorithm presented works also for non-deterministic plants, but we are unable to prove that at the moment. As noted in Chapter 3, though, either the plant or the supervisor is normally regarded as deterministic. The requirement that $S$ refines $P$ is natural. For the deterministic case, $L(S) \subseteq L(P)$ is agreed to be a reasonable assumption. In the same way we feel that $S$ refines $P$ is an equally reasonable assumption. As for accessibility, there is no real loss of generality at all in this. States not accessible in $S$ cannot concern the completeness of $S$, or its subprocesses, whatsoever. Note that $S$ is always a subprocess of itself, so that $S$ is also assumed to be accessible.

Note that, the three assumptions are essentially the same as those made in Section 6 of Wonham (1987), though here we allow the supervisor candidate to be non-deterministic. For practical purposes, all three assumptions are satisfied naturally.

Informally we have already mentioned the set of all complete subprocesses. The formal definition is made here.

**Definition 4.22 Set of All Complete Subprocesses**

The set of *all complete subprocesses* of a given process $S$, with respect to a process $P$ with uncontrollable event set $\Sigma_u$, is defined as

$$\mathcal{C}(S) = \{\, S' \in \mathcal{S}(S) |\, S' \textit{ complete w.r.t } P\}. \tag{4.27}$$

Completeness is a characterization of a process, by which we select elements from the set $\mathcal{S}(S)$ to compose the set $\mathcal{C}(S)$. In Section 4.1 we showed a number of interesting properties for sets of characterized subprocesses of a given process that are closed under subprocess union. To be able to use these in showing existence of a supremal complete subprocess, we have to show that the set of all complete subprocesses is closed under subprocess union.

Unfortunately, this is not the case, in general. This is illustrated by Example 4.4. It comes from the fact that in the union of two subprocesses, states that were not accessible in the synchronous composition of either of the subprocesses with the plant may become accessible by a string $s$ when synchronizing their union with the plant. It may be that the ready set of such a state does not include all uncontrollable events defined by the ready sets of the states reached by $s$ in the plant. If so, then the union of the subprocesses is not complete with respect to the plant and the uncontrollable events. Thus, we cannot show that the set of all complete subprocesses is closed under union, so that $\mathcal{C}(S)$ is not necessarily an upper semilattice. To use the upper semilattice properties to show existence of some supremal complete element, we have to add more constraints to the set we are searching, so as to guarantee that it is closed under union. This is where the three assumptions stated above come in. Unlike the characterizations of accessibility and coaccessibility, that only relied on the process itself, the completeness characterization relates the subprocesses to some other process, outside the set. Thus, it is impossible to regard the set of all complete subprocesses of $S$ without imposing some restriction relative to the plant. This restriction will be that $S$ refines $P$. This also means that all the subprocesses of $S$ refines $P$.

Completeness relates subsets of $Q_S$ and $E_S$ to subsets of $Q_P$ and $E_P$, namely those subsets that survive the synchronous composition $P \| S$. Thus, adding any number of non-accessible states and edges to $\sup \mathcal{C}(S)$ does not destroy its completeness. Of course, all of these must be present in $\sup \mathcal{C}(S)$. The problem is that $\sup \mathcal{C}(S)$ may not be unique. Another problem is that of computational complexity. To be sure to find the supremal element we do not want to compare $S$ and $P$ over and over again. Certainly not if all that it earns us is a lot of non-accessible states and edges that are of no importance for the closed-loop system. Because of this, we will only search for *accessible* and complete subprocesses of $S$. That is, we will search the supremal element of $\mathcal{C}(S) \bigcap \mathcal{A}(S) \equiv \mathcal{CA}(S)$.

Under the three assumptions above, we can now show the following lemma.

**Lemma 4.23** For a given accessible plant $P$ and an accessible specification process $S$ such that $S$ refines $P$, $\mathcal{CA}(S)$ is closed under arbitrary subprocess union. That is

$$Q, R \in \mathcal{CA}(S) \Rightarrow Q \bigcup R \in \mathcal{CA}(S). \tag{4.28}$$

**Proof.** The processes $Q$, $R$ and $Q \bigcup R$ are all subprocesses of $S$, and therefore they refine $S$. Since $S$ refines $P$ and refinement is transitive, see Lemma 2.41, $Q$, $R$ and

**Example 4.4 Subprocess Union not Complete**

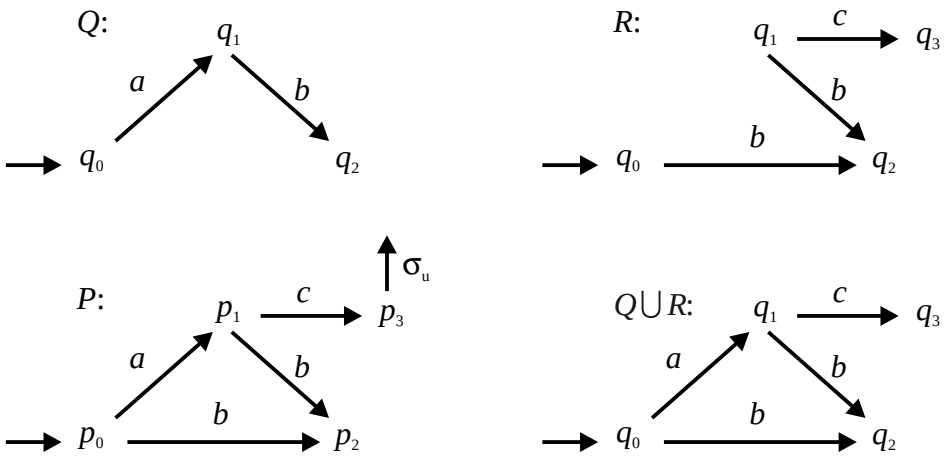



**Figure 4.7:** Illustration of the fact that the union of two complete subprocesses, with one non-accessible, is not necessarily complete.

In Figure 4.7 is shown two processes $Q$ and $R$, that we consider to be subprocesses of some process, say $S$, with $\Sigma_S = \Sigma_P$. Both $Q$ and $R$ are complete with respect to the plant $P$, with uncontrollable event $\Sigma_u = \{\sigma_u\}$. For $R$ the states $q_1$ and $q_3$ are not accessible, and thus $R$ is not accessible. However, in the union $Q \bigcup R$ these states become accessible, since $q_1$ is accessible in the process $Q$. The states $p_3 \in Q_P$ and $q_3 \in Q_{Q \bigcup R}$ are both reached by the same string, $s_1 s_3 \in L(P) \bigcap L(Q \bigcup R)$. Since $p_3$ has the uncontrollable event $\sigma_u$ in its ready set, and $q_3$ has not, it is obviously so that $q_3$ is an uncontrollable state of $Q \bigcup R$, and therefore $Q \bigcup R$ is not complete with respect to $P$ and $\Sigma_u$. Note though, that $q_3$ is not an uncontrollable state of $R$, since $q_3 \in Q_R$ is not accessible in $R$ and hence not in $P \| R$.

Example continued on next page

**Example 4.4 continued**

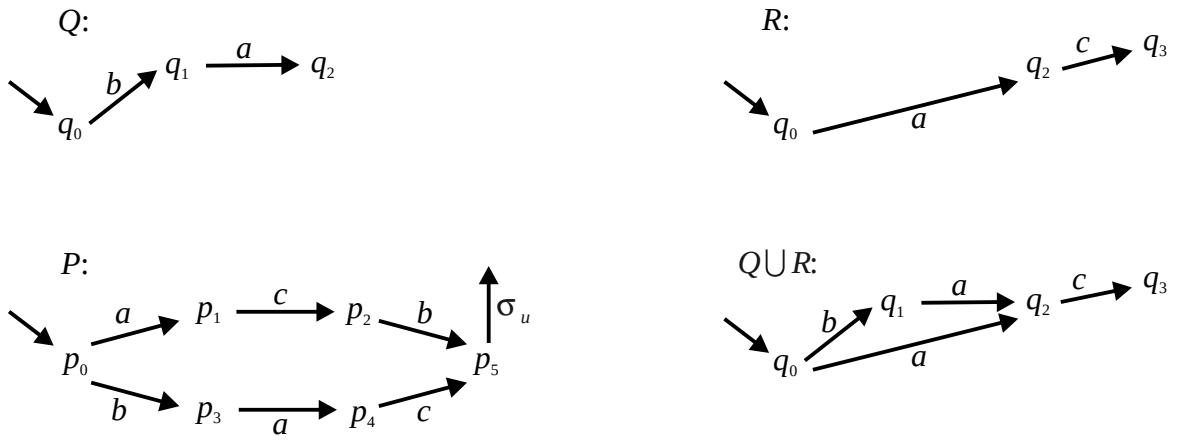



**Figure 4.8:** Illustration of the fact that the union of two deterministic and complete subprocesses is not necessarily complete, even though both are accessible.

In Figure 4.8 is shown two deterministic and accessible processes $Q$ and $R$, that we consider to be subprocesses of some process, say $S$, with $\Sigma_S = \Sigma_P$. Both $Q$ and $R$ are complete with respect to the plant $P$, and the uncontrollable event $\Sigma_u = \{\sigma_u\}$. Since both subprocesses are accessible, the union $Q \bigcup R$ is also accessible. However, even though both subprocesses are complete, their union is not. After the string $bac \in L(P) \bigcap L(Q \bigcup R)$ the plant is in state $p_5$ while $Q \bigcup R$ is in $q_3$. The uncontrollable event $\sigma_u = \text{out}(p_5)$, while $\sigma_u \notin \text{out}(q_3)$, and therefore $Q \bigcup R$ is not complete. This follows from the fact that the union of the language of two subprocesses is a subset of the language of their union. That is, $L(Q) \bigcup L(R) = \left\{\overline{ac}, \overline{ba}\right\} \subseteq L(Q \bigcup R) = \left\{\overline{ac}, \overline{bac}\right\}$, where the overbar represent prefix closure.

Note also that, though both $Q$ and $R$ refine $P$, their union does not. This is because the function defining refinement is different for $Q$ and $R$. We have that $f_Q(q_2) = p_4$ and $f_R(q_2) = p_2$. Therefore, $f_{Q \bigcup R}(q_2)$ is not unique, either $p_4$ or $p_2$, and thus it is easy to see that there exists no function such that $Q \bigcup R$ refines $P$.

■

$Q \bigcup R$ also refine $P$. By Theorem 2.48 this means that $P \| Q = Q$, $P \| R = R$ and $P \| (Q \bigcup R) = Q \bigcup R$, since $P$ is deterministic. Furthermore, when $Q$ and $R$ are accessible, so is $Q \bigcup R$.

Assume that there exists a trace $t \in tr(Q \bigcup R)$ such that $s = \text{label}(t) \in L(P \| (Q \bigcup R))$. Assume further that $out(\delta(P, s)) \bigcap \Sigma_u \not\subseteq out(\text{last}(t))$, that is, the state $q = \text{last}(t)$ is uncontrollable and hence $Q \bigcup R$ non-complete. Say that $q \in Q_R$. Since $P \| R = R$ and $R$ is accessible, there exists a trace $t_R \in tr(R) \subseteq tr(Q \bigcup R)$ such that $q = \text{last}(t_R)$ and $s_R = \text{label}(t_R) \in L(P \| R) \subseteq L(P \| (Q \bigcup R))$. Denote by $t_P$ the trace of $tr(P)$ such that $\text{label}(t_P) = s = \text{label}(t)$, and by $t'_P$ the trace of $tr(P)$ such that $\text{label}(t'_P) = s_R = \text{label}(t_R)$. Since $P$ is deterministic there can only exist one of each. Then, since $t$ and $t_R$ both belong to $tr(Q \bigcup R)$, $P$ is deterministic, $Q \bigcup R$ refines $P$ and $t \; rel_{Q \bigcup R} \; t_R$, $\text{last}(t_P) = \text{last}(t'_P)$ so that $\langle last(t_P), last(t) \rangle = last(t_P \, \| t \,) = last(t'_P \, \| t_R \,) = \langle last(t'_P), last(t_R) \rangle$.

Since $R$ is complete, $out(last(t_P)) \bigcap \Sigma_u \subseteq out(last(t_R))$ which contradicts that $q$ is uncontrollable. Therefore, $\mathcal{CA}(S)$ is closed under union when $S$ refines $P$. ■

**Remark.** Here we have denoted by $t_P \, \| t$ and $t'_P \, \| t_R$ the traces of $P \| (Q \bigcup R)$ resulting from the synchronization. Obviously, $P$ being deterministic is essential here, otherwise there may exist several non-related traces with the same label $s$. Knowing that $R$ is complete, is then no guarantee for $Q \bigcup R$ being complete. Note also that the proof holds equally for the case that $q \in Q_Q$.

The null process, $\emptyset_S$, is accessible, refines any process and is complete with respect to any plant. Thus, $\emptyset_S \in \mathcal{CA}(S)$, so that the characterization $\mathcal{CA}(\cdot)$ satisfies the assumption made in Section 4.1. □

From Section 4.1 we know that Lemma 4.23 means that $\mathcal{CA}(S)$ is an upper semilattice under the given conditions. Therefore, there exists a unique supremal element of $\mathcal{CA}(S)$, the union over all elements of $\mathcal{CA}(S)$.

**Theorem 4.24** The *supremal accessible and complete subprocess* of $S$ (with respect to $P$ and $\Sigma_u$), when $S$ refines $P$, $\sup \mathcal{CA}(S)$, exists and is unique.

**Proof.** Since $\mathcal{CA}(S)$ is an upper semilattice it is well-known that $\sup \mathcal{CA}(S)$ exists and is unique. ■

**Remark.** Note though, that $\sup \mathcal{CA}(S)$ may be the null process, as could be the case when all events of $P$ are uncontrollable. □

To synthesize $\sup \mathcal{CA}(S)$, we could iteratively compare $P$ and $S$, applying the test for completeness and removing the uncontrollable edges on each iteration. However, a more efficient algorithm, comparing $P$ and $S$ only once and then operating on $S$ alone, will be described here. The following algorithm generates a complete subprocess of $S$, which we will call $S_c$.

**Example 4.5 Union of Complete Subprocesses**

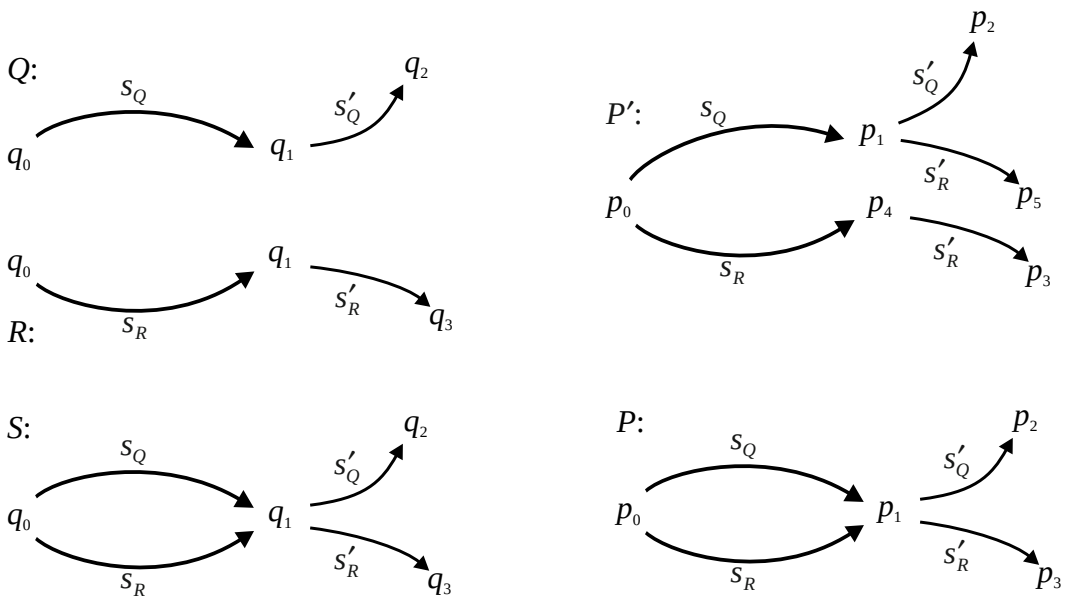



**Figure 4.9:** A specification $S$, pertaining to two plants, $P$ and $P'$ such that $S$ refines $P$ but not $P'$. $Q$ and $R$ are two complete subprocesses of $S$.

The proof for the fact that $\mathcal{CA}(S)$ is closed under union, is illustrated in Figure 4.9. The specification $S$ refines the plant $P$, but not $P'$. Therefore, in $P'$ $s_Q$ and $s_R$ can reach different states, even though they reach the same states in $S$. When $S$ refines $P$ we know that this is not the case. In this specific example, $S$ is a subprocess of $P$.

Two subprocesses of $S$, $Q$ and $R$ are shown to the top left in Figure 4.9. We assume these to be complete with respect to $P$, even though no uncontrollable events are shown. $S$ is not necessarily complete, though in this case it is. If $Q$ and $R$ were to have only initial states in common (or none) then there would be no problem, since then $tr(Q \bigcup R) = tr(Q) \bigcup tr(R)$. However, in this case the state $q_1$ is in both state-spaces. Therefore, there arises a new trace in $tr(Q \bigcup R)$, namely $t_Q t'_R = (q_0, s_Q, q_1)(q_1, s'_R, q_2)$. A string with this label also exists in the plant, so that this trace will survive under synchronization with the plant. However, if this trace should make $Q \bigcup R$ non-complete, then also $R$ must be non-complete.

Example continued on next page

**Example 4.5 continued**

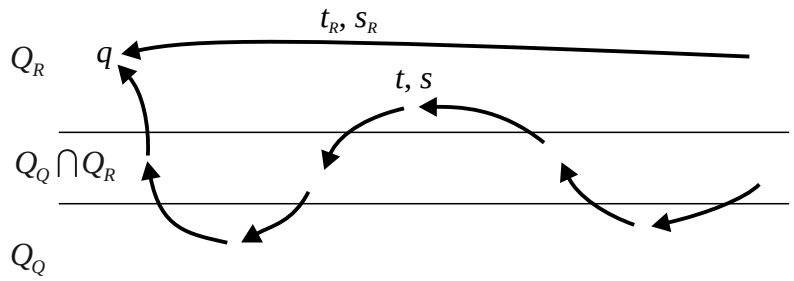



**Figure 4.10:** Illustration of how jumping between the state-spaces of $Q$ and $R$ is possible in $Q \bigcup R$.

In Figure 4.9 the proof of Lemma 4.23 is illustrated for one "step" only. However, in $Q \bigcup R$ we may jump back and forth between $Q_Q$ and $Q_P$, as shown in Figure 4.10. Nonetheless, for *all* states it holds that they are accessible, and each corresponds to one distinct state of $Q_P$, when $S$ refines $P$. Note that the in initial states $q_0$, $q_0'$ and $q_0''$ may be the same state.

■

**Algorithm 4.25 Supremal Complete Subprocess**

Define an operator $\Phi(Q, Q_i) : 2^{Q_S} \times 2^{Q_S} \rightarrow 2^{Q_S}$ on a state-space $Q_S$ of a transition machine $S$ such that

$$\begin{aligned} &\Phi(Q, Q_i) = \\ &\{q \in (Q_S - Q) | \, \exists (q, \sigma_u, q') \in E_S \;\; \text{such that } \sigma_u \in \Sigma_u \wedge q' \in Q_i\} \end{aligned} \tag{4.29}$$

1. Compare $P$ and $S$, set $Q_0 \leftarrow \{q \in Q_s \, | q \;\; \text{is an uncontrollable state}\}$
2. Iterate $Q_{i+1} \leftarrow \Phi(\bigcup_{j=0}^{i} Q_j, Q_i)$ until $Q_{i+1} = \emptyset$
3. $Q_{uc} = \bigcup_{j=0}^{i} Q_j$ and then $Q_{S_c} = Q_S - Q_{uc}$ and $E_c = E_S - \{e \in E_S \, | last(e) \in Q_{uc}\}$

**Remark.** An *uncontrollable state*, is a state $q \in Q_S$ reached by a string $s \in L(P) \bigcap L(S)$ such that there exists $p \in Q_P$ with $\text{out}(p) \bigcap \Sigma_u \not\subseteq \text{out}(q)$. Furthermore, by an *uncontrollable edge* (controllable edge), we mean an edge labeled by an uncontrollable (controllable) event.

Note that only transitions are removed from $S$ to generate $S_c$. The state-space of $S_c$ is equal to the state-space of $S$; an (uninteresting) technicality we employ just to simplify the proof of this algorithm.

□

The operator $\Phi$ picks out those states of $Q_S$ not in $\bigcup_{j=0}^{i} Q_j$ from which uncontrollable edges lead into $Q_i$. Each $Q_i$ can thus be seen as a *forbidden* subspace of the state-space $Q_S$.

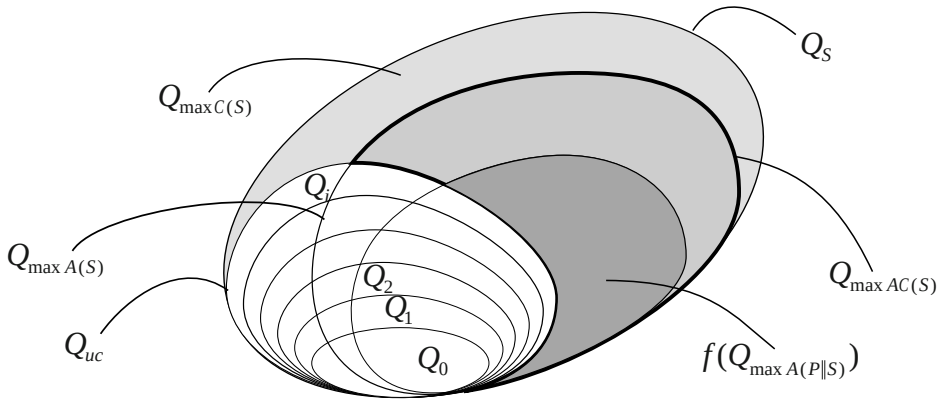



**Figure 4.11:** A general description of the various partitions of the state-space of an arbitrary specification $S$ not necessarily refining $P$.

For $S$ to be complete it cannot be allowed to enter any forbidden subspace, since then, either the state reached is uncontrollable, or the state reached can reach an uncontrollable state by one or more uncontrollable edges. In Algorithm 4.25, $\Phi$ is applied iteratively to the states not already in the forbidden subspace, calculating a new forbidden subspace, extending the previous one. The union over all forbidden subspaces is then *the* forbidden subspace of $Q_S$, which $S$ must be prevented from reaching. By removing those edges leading into the forbidden subspace this is assured. Since the iteration goes on until no more uncontrollable edges reach into the previous subspace, it is guaranteed that $S$ can be prevented from reaching the forbidden subspace by removing controllable edges, only. See also Example 4.6 on page 133.

**Lemma 4.26** Algorithm 4.25 terminates in at most $|Q_S|$ iterations.

**Proof.** All of the $Q_i$ are disjoint, since $q \in Q_{i+1} \Leftrightarrow q \in Q_S \wedge q \notin Q_0 \bigcup \ldots \bigcup Q_i$. Therefore (for $n = |Q_S|$) $Q_n = Q_S - \bigcup_{j=0}^{n-1} Q_j = \emptyset$. ∎

**Remark.** Note that termination after $|Q_S|$ iterations is a worst case scenario with all $Q_i$ singleton. Normally the algorithm will stop before the entire state-space is exhausted, since once $Q_i = \emptyset$ $Q_{i+k} = \emptyset$ for $k = 1, 2, \ldots$ □

In Figure 4.11 is given a general description of the various partitions of the state-space of an arbitrary specification, not necessarily refining $P$. When $S$ is accessible and refines $P$, then its state-space is given by the inner boundary, surrounding the dark gray area. The extended set of uncontrollable states, is represented by the white area, $Q_{uc} = Q_0 \bigcup Q_1 \bigcup \ldots$ The remaining states $Q_S - Q_{uc},$, represented by the entire gray area, are controllable. The gray area within the thick boundary is the states of $\sup \mathcal{CA}(S)$, while the dark gray area is the states that survive under synchronization, $f(Q_{\sup \mathcal{A}(P\|S)}) \subseteq Q_S$. Note that the state-space of $\sup \mathcal{C}(S)$ (assuming it exists) includes everything but the intersection $Q_{uc} \bigcap f(Q_{\sup \mathcal{A}(P\|S)})$.

Naturally, $S_c$ as given by Algorithm 4.25 is not necessarily accessible. Therefore we have to calculate the supremal accessible subprocess of $S_c$, that is, $\sup \mathcal{A}(S_c)$. Obviously, this does not destroy completeness, since it merely removes states and edges that are non-accessible due to the removal of $Q_{uc}$ and its associated edges. $\sup \mathcal{A}(S_c)$ is both accessible and complete.

Now we must also show that the accessible and complete subprocess $\sup \mathcal{A}(S_c) \leq S$ found by Algorithm 4.25 is the *supremal* accessible and complete subprocess. That is, there does not exist a transition machine $S' \leq S$ such that $S'$ is accessible and complete and $\sup \mathcal{A}(S_c) \leq S'$. This is shown by the following lemma.

**Lemma 4.27** Algorithm 4.25 finds the supremal accessible and complete subprocess of $S$.

**Proof.** Consider how Algorithm 4.25 works. $Q_0$ contains the uncontrollable states, that is, states that can be reached by strings of the closed-loop system such that $P$ can generate an uncontrollable event which $S$ cannot follow. Since $S$ is accessible and $L(S) \subseteq L(P)$, $Q_0$ are all accessible states. By definition, for $S$ to be complete it must be such that it prevents the closed loop system from ever reaching these states. Formally,

$$Q_0 = \{q \in Q_S \,|\exists s \in L(P \parallel S)\ \langle p, q\rangle \in \delta(P \parallel S, s)\ out(p) \bigcap \Sigma_u \not\subseteq out(q)\} \tag{4.30}$$

This is straightforwardly given by the inverse of (3.23), with $P \parallel S$ instead of regarding $P$ and $S$ separately. Next, $Q_1$ is generated by selecting those states which lead into $Q_0$ by uncontrollable events. Again, $Q_1$ only contains accessible states. That is,

$$Q_1 = \{q \in Q_S - Q_0 \,|\exists (q, \sigma_u, q') \in E_S\ \sigma_u \in \Sigma_u \wedge q' \in Q_0\}. \tag{4.31}$$

This is equivalent to

$$Q_1 = \{q \in Q_S - Q_0 \,|\exists \sigma_u \in out(q) \bigcap \Sigma_u\ \delta_S(q, \sigma_u) \bigcap Q_0 \neq \emptyset\}. \tag{4.32}$$

For $Q_2$ we similarly have that $Q_2$ contains those states which lead us into $Q_1$. That is

$$Q_2 = \{q \in Q_S - (Q_0 \bigcup Q_1) \,|\exists \sigma_u \in out(q) \bigcap \Sigma_u\ \delta_S(q, \sigma_u) \bigcap Q_1 \neq \emptyset\} \tag{4.33}$$

For $Q_1$, by (4.32) a single uncontrollable event takes us into $Q_0$. For $Q_2$, a single uncontrollable event takes us into $Q_1$, from where a single uncontrollable event takes us into $Q_0$. That is, from $Q_1$ and $Q_2$ a string of length 1 and length 2, respectively, of uncontrollable events take us into $Q_0$. Naturally, from $Q_i$, a string of length $i$ of uncontrollable events takes us into $Q_0$. Of course, this includes the string of length zero, the null string $\varepsilon$, taking us from $Q_0$ to $Q_0$ itself.

Thus, for the union over all $Q_i$ we have that these are the states from which strings of uncontrollable events of any length less than $i + 1$ lead to $Q_0$. The subprocess of $S$ with the transitions to these states removed is $S_c$. It is clear that $\sup \mathcal{A}(S_c)$ is complete, that is $\sup \mathcal{A}(S_c) \in \mathcal{CA}(S)$ and $\sup \mathcal{A}(S_c) \leq \sup \mathcal{CA}(S)$.

Assume now that $\sup \mathcal{CA}(S) \not\leq \sup \mathcal{A}(S_c)$. Then $Q_{\sup \mathcal{CA}(S)} \not\subseteq Q_{\sup \mathcal{A}(S_c)}$ or $E_{\sup \mathcal{CA}(S)} \not\subseteq E_{\sup \mathcal{A}(S_c)}$. Since both $\sup \mathcal{CA}(S)$ and $\sup \mathcal{A}(S_c)$ are accessible, their edge-sets contain all

states. Therefore, when $\sup \mathcal{CA}(S) \not\leq \sup \mathcal{A}(S_c)$, there exists an edge $e' \in E_{\sup \mathcal{CA}(S)}$ such that $e' \notin E_{\sup \mathcal{A}(S_c)}$. Of course, $e'$ is not accessible or $last(e') \in Q_{uc}$, otherwise $e'$would not have been removed from $E_S$ in forming $E_{\sup \mathcal{A}(S_c)}$ Thus, $\sup \mathcal{CA}(S)$ is not complete or not accessible. This is of course a contradiction, saying that if $\sup \mathcal{CA}(S) \neq \sup \mathcal{A}(S_c)$ then $\sup \mathcal{CA}(S) \notin \mathcal{CA}(S)$, and thus $\sup \mathcal{CA}(S) = \sup \mathcal{A}(S_c)$. ■

**Remark.** Note that the proof relies on $L(S) \subseteq L(P)$. Otherwise, some accessible edges pertaining to strings in $L(S)$ not in $L(P)$ could be allowed in $\sup \mathcal{CA}(S)$, since these strings cannot be present in the closed-loop system. The algorithm, however, makes no such distinction. $L(S) \subseteq L(P)$ makes sure that such edges do not arise. □

## 4.4 The Supremal Complete and Trim Subprocess

We know that the sets $\mathcal{CA}(S)$ and $\bar{\mathcal{A}}(S)$ are closed under subprocess union. By Lemma 4.6 we then also know that $\mathcal{CA}(S) \bigcap \bar{\mathcal{A}}(S)$ is closed under subprocess union, so that a unique supremal element $\sup \left[\mathcal{CA}(S) \bigcap \bar{\mathcal{A}}(S)\right]$ exists. The problem is finding that element algorithmically. One way is to generate the sets $\mathcal{CA}(S)$ and $\bar{\mathcal{A}}(S)$, and then take the supremal element of $\mathcal{CA} \bigcap \bar{\mathcal{A}}(S)$. Though, formally no problem, this approach is not usable in practice, due to the large amount of subprocesses that has to be generated for a realistic implementation.

We want to generate as few subprocesses as possible while still being guaranteed that we can find the supremal element. The following algorithm uses Algorithm 4.25 and Algorithm 4.21 to iteratively calculate the supremal complete and trim subprocess until a fixpoint is reached. Thus, we will only generate as many subprocesses as is required to find the one that is trim, complete and supremal. Naturally, under the assumptions that the plant is deterministic and the specification refines the plant.

**Algorithm 4.28 Supremal Complete and Trim Subprocess**

Using Algorithm 4.25 and Algorithm 4.21 above, calculating the supremal complete and supremal trim subprocess, we iterate through the following steps.

1. Set $S_0 \leftarrow \sup \mathcal{CA}(S)$
2. Calculate $S_{i+1} \leftarrow \sup \bar{\mathcal{A}}(S_i)$ if $S_{i+1} = S_i$ terminate, else
3. Calculate $S_{i+2} \leftarrow \sup \mathcal{CA}(S_{i+1})$ if $S_{i+2} = S_{i+1}$ terminate, else
4. Goto 2

**Remark.** Initially the supremal complete and accessible subprocess is calculated, then the supremal coaccessible subprocess of this. If a fixpoint is not reached, the supremal complete and accessible subprocess of the resulting transition machine is calculated, and so on until termination. See also Example 4.6. □

**Lemma 4.29** Algorithm 4.28 terminates in at most $\frac{n-1}{2}$ number of iterations, for $n = |Q_S|$.

**Proof.** In the worst case, we iterate until $\emptyset_S$ and only one state for each calculation of $\sup\mathcal{CA}(S_j)$ or $\sup\bar{\mathcal{A}}(S_i)$ is removed by Algorithm 4.25 and Algorithm 4.21, respectively. Thus, two states per iteration in Algorithm 4.28 are removed. On the initial step one state is removed in the worst case. With $|Q_S| = n$ we have that $1 + 2i = n$, for number of iterations i. This means that the supremal number of iterations can be $i = \frac{n-1}{2}$. ■

**Remark.** Starting with $|Q_S| = n$, $S_0$ has $n-1$ states. Calculating $S_1 = \sup\bar{\mathcal{A}}(S_0)$ requires $n-1$ operations, and $S_1$ has $n-2$ states. Calculating $S_2 = \sup\mathcal{CA}(S_1)$ then requires $n-2$ operations and $S_2$ has $n-3$ states, and so on. It is obvious that the total number of operations, that is, the total number of states we have to check, in the worst case, is $n + (n-1) + \ldots + (n-n+1) = \frac{n(n+1)}{2}$. □

Does Algorithm 4.28 really generate the *supremal complete and trim* subprocess? The answer has in fact already been given in Theorem 4.14, but the following theorem proves this for Algorithm 4.28 specifically.

**Theorem 4.30** Algorithm 4.28 generates the *supremal complete and trim* subprocess.

**Proof.** The proof follows from Theorem 4.14. We know from the fact that $\mathcal{CA}(S)$ and $\bar{\mathcal{A}}(S)$ are closed under union and from Lemma 4.6 that $\mathcal{CA}(S) \bigcap \bar{\mathcal{A}}(S)$ is closed under union. What Algorithm 4.28 does is in fact implement the supremal operator, defined by Definition 4.10. To see this, note that the iteration of Algorithm 4.28 first calculates $\sup\mathcal{CA}(S)$, then $\sup\bar{\mathcal{A}}(\sup\mathcal{CA}(S))$ and then $\sup\mathcal{CA}(\sup\bar{\mathcal{A}}(\sup\mathcal{CA}(S)))$, and so on. Thus, Algorithm 4.28 implements the operator $\sup\bar{\mathcal{A}}\mathcal{CA}_n(S)$.

The iteration halts when $\sup\bar{\mathcal{A}}\mathcal{CA}_i(S) = \sup\bar{\mathcal{A}}\mathcal{CA}_{i+1}(S)$, and by Theorem 4.14 we then know that this is indeed the supremal element $\sup\left[\mathcal{CA}(S) \bigcap \bar{\mathcal{A}}(S)\right]$. That is, the *supremal complete and trim subprocess* of $S$, $\sup\left[\mathcal{C} \bigcap \mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S)\right]$. ■

An algorithm for finding the supremal controllable sublanguage of a given language is given by Wonham (1987). This algorithm is of order $m^2n^2$, according to Kumar (1991) Remark 3.12, where $m$ and $n$ are the number of states in the minimal state-machine realizations of the languages $L(P)$ and $K$, respectively. Kumar (1991) gives an algorithm of order $mn$ operating on automata. However, this algorithm is only described for deterministic state-machines and does not guarantee a trim closed-loop system.

In Kumar (1995) is shown a similar application of lattice theory to synthesis of supervisors. However, Kumar (1995) considers deterministic automata, and so models a discrete event process by its language. Therefore, *languages* are ordered into lattice structures, whereas we order automata, which are allowed to be non-deterministic. Kumar (1995) shows that a non-deterministic process can always be transformed into a deterministic process with the same language. A fact also shown by Hopcroft (1979). This is irrelevant to the approach taken in this work, however. To us the states have significant meaning. Even if two states can be reached by the same string, they do not (necessarily) convey

the same information. Typically, the states distinguish between *which specific product* has undergone what sequence of operation. This information is lost in the minimal, deterministic automaton, where a state only says that *some product* has been through the sequence of the events leading to this state. This is not detailed enough for us, as was shown in Example 2.3.

Algorithm 4.28 presented in this section is similar to Algorithm 3.1 of Kumar (1995), though there it is presented without proof. (The proof is left as an exercise for the reader.) With the operators defined in Algorithms 4.25, 4.16 and 4.19, we feel that our versions are more compact and concise.

## 4.5 Chapter Summary

In this chapter we have shown that transition machines can be ordered in lattice structures according to the subprocess relation. Furthermore, sets of subprocesses can be distinguished by certain characterizations, mainly the set of all complete and accessible and the set of all trim subprocesses. These sets can then be ordered in upper semilattice structures, so that the supremal element of the respective sets exists. For the set of all complete and accessible subprocesses, we also have to add the constraint that the generating element refines the plant. We have shown how to find these supremal elements, and given algorithms for automatic synthesis. We have also shown how these algorithms can be used to find the supremal complete and trim subprocess of a given transition machine. This is done iteratively, no general closed form expressions exist. Given a transition machine as a specification for the desired behavior of a plant, the supremal complete and trim subprocess of that specification is, of course, the minimally restrictive supervisor guaranteeing that the closed-loop system is trim and always behaves within the boundaries set by the specification.

Some of the lattice results given are well-known and shown by Tremblay (1987), for instance. Other results for the specific application of calculating a complete and trim supervisor are new. The definition of the *supremal operator* to calculate the supremal element of the intersection of two upper semilattices, when it exists, is new, as far as we know. The algorithms given for calculating complete, accessible and coaccessible subprocesses are not new, however. Eilenberg (1974) also shows the algorithm for accessibilty and coaccessibility, while the algorithm for completeness is well-known within the supervisory control theory. See Kumar (1991), for instance. What is new, is the application of the algorithms to non-deterministic systems, and the short and concise notation resulting from the definition of the various operators.

In the next chapter we will elaborate on aspects of specifications, which can be given with various intentions, such as forbidden states and connected events. Enforcing these specifications generates a subprocess of the original specification from which the supervisor subprocess is synthesized. Since we know that a subprocess refines all processes that its superprocess refines, enforcing such specifications is legal; we know that such manipulations are allowed within the theory presented in the previous chapters. In fact, any operation that preserves the refinement property can be applied, which means that we can also specifiy forbidden strings and add extra marking, if necessary.

**Example 4.6 Supervisor Calculation**

In this example the process $S$ is a supervisor candidate for the plant $P$, and the supremal complete subprocess of $S$, with respect to $P$ will be generated. These processes are chosen so as to best illustrate the described characteristics of this calculation, yet at a manageable size. We will not even try to give any physical interpretation of the shown transition machines. In Chapter 6 examples with relevant practical aspects will be described.

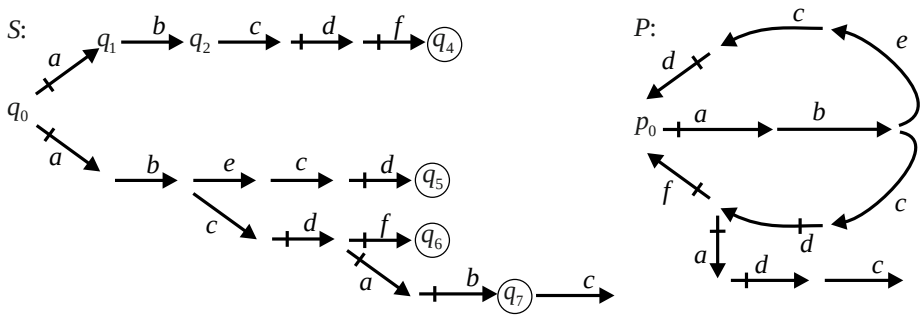



**Figure 4.12:** The supervisor candidate $S$ and the plant $P$. The uncontrollable events are $\Sigma_u = \{b, c, e\}$. The initial states are $I_P = \{p_0\}$ and $I_S = \{q_0\}$. The encircled states are the marked states of $S$, that is $M_S = \{q_4, q_5, q_6, q_7\}$. Notice that only states of importance have been named.

In Figure 4.12 the specification and the plant is shown. It is obvious that both processes are accessible, $P$ is deterministic and $S$ refines $P$. Therefore, we can use Algorithm 4.25. However, it is also obvious that $S$ is not trace nonblocking, since a terminal state is not marked.

Comparing $S$ and $P$, the algorithm sets $Q_0 = \{q_2\}$, since after the string $ab$, $P$ can execute the events $c$ and $e$, both of which are uncontrollable. However, after $ab$ in the state $q_2$, $S$ can only follow event $c$, and thus $S$ is not conforming. Hence $S$ is not complete with respect to the plant. Note though that $L(S)$ is controllable. The state $q_3$ is thus an uncontrollable state, and it is the only uncontrollable state, as is easily verified.

In calculating $Q_1$, Algorithm 4.25 applies the $\Phi$ operator, with $Q_0$ as both arguments. Thus, states of $Q_S$ not in $Q_0$ are checked to see if any exist such that they reach into $Q_0$ via an uncontrollable event. In this case, $q_1$ is the only state that reaches $q_2$ by an uncontrollable event, $b$. Thus, $Q_1 = \{q_2\}$.

Next, the operation $\Phi(Q_0 \bigcup Q_1, Q_1) = \Phi(\{q_2, q_3\}, \{q_2\})$ is performed. However, no states of $Q_S - Q_0 \bigcup Q_1$ reach into $Q_1$ by uncontrollable events, so $Q_2 = \emptyset$, and the iteration terminates. Now all those edges reaching some state of $Q_0 \bigcup Q_1$ is removed. In this case the edges $(q_1, b, q_2)$ and $(q_0, a, q_1)$ are removed. Note that the transition machine is "pruned" at controllable events only.

The resulting transition machine is shown in Figure 4.13. It is clear that the resulting process is complete with respect to the plant. However, it is also clear that it is not trim. Even if it would have been trim to start with, by removing the mentioned edges, the marked state $q_4$ is no longer reachable from the initial state $q_0$.

Example continued on next page

**Example 4.6 continued**

Next we show how Algorithm 4.21 calculates the supremal trim subprocess of the result obtained from calculating $S_c$ of Figure 4.12.

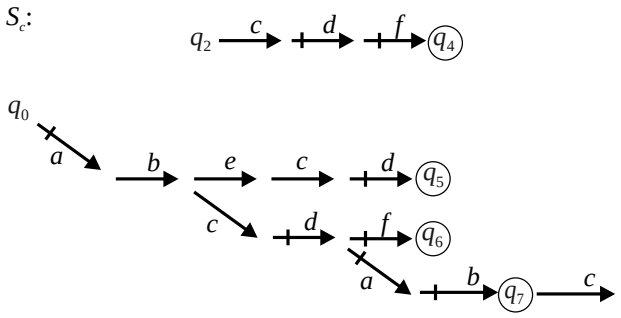



**Figure 4.13:** The complete subprocess $S_c \leq S$, as calculated by Algorithm 4.25.

In calculating the supremal trim subprocess of Figure 4.13, Algorithm 4.21 uses Algorithm 4.19 to calculate the subprocess $\sup \bar{\mathcal{A}}(\sup \mathcal{A}(S_c))$. First the non-accessible states are removed. Clearly, $q_2$, the edges reaching $q_4$ are non-accessible. Removing these generates $\sup \mathcal{CA}(S)$.

However, $\sup \mathcal{CA}(S)$ is accessible but not coaccessible. It is not coaccessible, since the terminating state reached by $c$ from $q_7$ cannot reach any of the marked states, $\{q_4, q_5, q_6, q_7\}$. Thus, we will have to calculate the supremal coaccessible subprocess of $\sup \mathcal{CA}(S)$, that is $\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(S))$. This removes the $c$-transition from $q_7$. The resulting process, $\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(S))$ is shown to the left in Figure 4.14. This process is both accessible and coaccessible, but it is no longer complete, even though it was complete before it was made coaccessible.

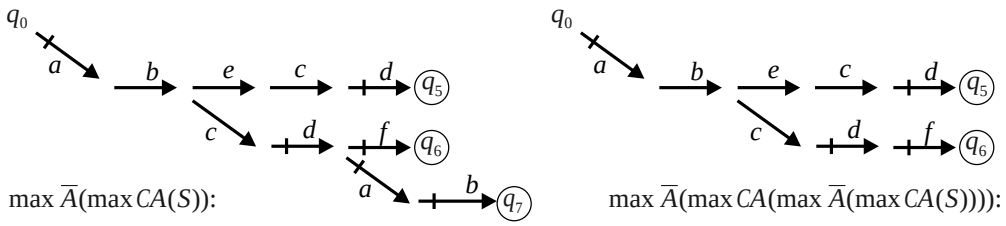



**Figure 4.14:** The resulting processes of the respective calculations $\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(S))$ and $\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(S))))$. Note that $\sup \bar{\mathcal{A}}(\sup \mathcal{CA}(S))$ is no longer complete, even though it was complete before it was made coaccessible.

Therefore, the calculation of the supremal complete and accessible subprocess has to be carried out again. As it turns out, this process is not coaccessible, so the supremal coaccessible part of it has to be calculated. Finally, we arrive at a process that is complete, accessible and coaccessible. This process is shown to the right in Figure 4.14. Note that, though in this example the resulting complete and trim supervisor is deterministic, this is not generally the case.

■

# Chapter 5

# Control of Discrete Event Fabrication Processes

In this chapter we aim to put the supervisor synthesis procedure into perspective. Implementing a control system is not just a matter of synthesizing a supervisor. There are numerous other aspects that have to be considered, and inconsistencies between these and the supervisory control theory have to be resolved.

Even though the main concern of this thesis has hitherto been on supervisor synthesis, given a plant model and product specifications, it has been inevitable not to investigate the aspects of plant and product modeling. In fact, most of the problem statements and results in the previous chapters originated from research of applications of object-oriented design methods to the implementation of control systems for flexible fabrication processes[1]. This work has since been continued by a cross-departmental academic research team, in conjunction with a number of industrial participants. Here we present the early results already presented in our previous works shown on page ii, though modified to fit the terminology and proposed structure of the continued work in its current status.

We begin with some aspects of modeling discrete event fabrication processes, relevant to the implementation of control-systems, as well as for actual control of such processes. We argue that an object-oriented design approach facilitates both the implementation and the flexibility, due to its inherent feature of separating control of the individual physical subsystems from control of the system as a whole. A description of the object-oriented modeling approach follows. This is then followed by a discussion of high-level product descriptions in such systems. Then we discuss a specific aspect of event synchronization that is needed as a result of using reusable autonomous resources to model the plant, *event connection* defined in Section 5.4.2. Next we describe how the individual product routes are to be joined to describe a global specification on the behavior of the plant. By interleaving the product routes in their Petri net form we model the sharing of the resources among the products and synchronize with the plant model. It is from this specification that we generate the supervisor. Finally we summarize the approach, and apply the theory presented in the previous chapters.

[1] Here we use the term *fabrication process* to mean, not only discrete event manufacturing systems, but also batch processes and assembly systems.

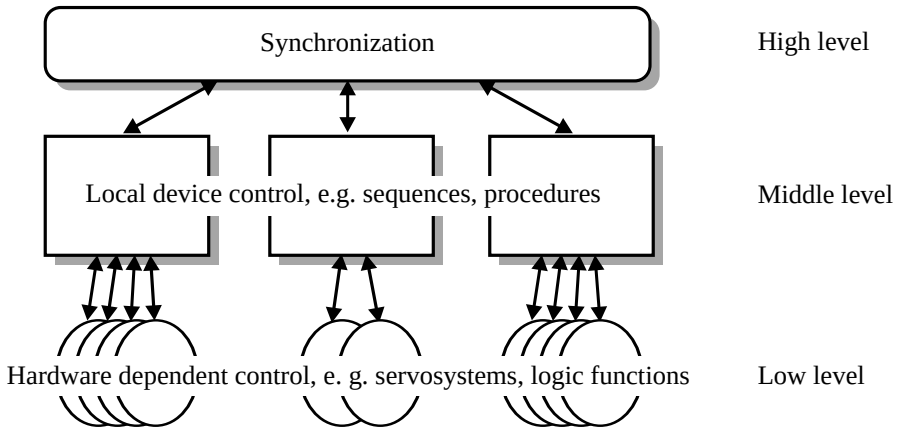



**Figure 5.1:** Three levels of fabrication process control.

## 5.1 Introduction

Control of manufacturing systems can be applied on several different levels. We will identify three of these, and simply call them the *low*, *middle* and *high* level. See Figure 5.1. On the low level the emphasis of control is on the implementation of certain behavior of each physical device involved in the manufacturing process. At this level very simple primitives, like Boolean functions and feedback control-loops, make sense. Each device is controlled separately and close to the hardware. Some local control mechanism regulates the flow of energy and other resources of the device in accordance with a given specification in order to accomplish a given operation. For instance, to control the movement of a robot arm at this level would involve controlling the position, speed and acceleration of each individual joint.

On the middle level the device is viewed as a more unified entity, offering particular services to be invoked, with the lower level implementation being hidden from the user. At this level control over the device is more a question of sequencing, allowing for logical-branch primitives. Complex actions can thus be described in a more feasible way, building on the capabilities of the lower level. The movement of a robot arm can be described in terms of moving the end effector, the individual joints are of no concern at this level. Still, the control is local, focused on each device, and hence makes no provision for expressing the interworking between the devices that must exist for profitable work to be accomplished.

On the high level the emphasis is on the problem of synchronizing the individual devices. In order for the manufacturing of products to take place in an ordered way, the controller has to ensure that some actions are taken before others, and that other actions are not carried out if various conditions are not fulfilled, etc. A machine can obviously not be loaded unless the previous workpiece has been unloaded, a robot cannot fetch a new item while it has not disposed of the current one, and so on. Obviously, this calls for some kind of sequential control to guarantee the orderly flow of events. This thesis shows that the supervisory control can provide the means to generate such control laws for discrete event fabrication processes.

The low and middle levels have traditionally been the realm of Control Engineering,

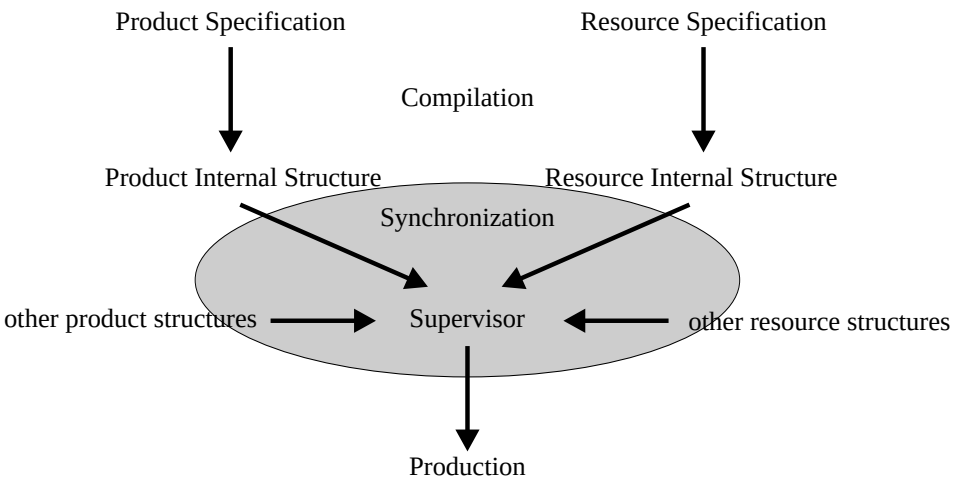



**Figure 5.2:** Synthesization structure of a control system for a flexible fabrication process. In this thesis we have in the previous chapters mainly been concerned with the part within the gray oval; generating a supervisor given product specifications and resource models.

with the high level being seen as subdivided between Computer Science and Operations Research. However, with the supervisory control theory, especially in the input/output interpretation, the problem on the high level can be stated as "given a dynamic plant, find a controlling entity to keep the output of the plant within some given specification". This statement could just as well describe the traditional problem of control engineering; the only difference between the types of control is the character of the available control actions and plant output.

The emphasis of the work presented in this thesis is on the global control software at the high level, not on the local control of individual devices. Our aim is to facilitate true flexibility in fabrication processes by separating the global control at the high level from the lower level specific control. Also, we want the system to be able to incorporate new products and new equipment with minor or no reprogramming of the control system. This is achieved by using a *modular control system* where the modules are as independent of each other as possible, with control laws synthesized from suitable models of the products and the machining resources.

As mentioned in Chapter 1, one of the main reasons for fabrication processes to be inflexible, despite the fact that their components are themselves highly flexible, is that the control system is heavily influenced by the product routes relevant at the time of implementation. A way to remedy this is to impose a modular control system with a strong separation between the modules. The stronger the separation between two modules, the more we are guaranteed that a change in one of them will not affect the other.

One way to achieve a strong separation between two modules of a control system is to implement one without regard for the implementation of the other. For instance, we should be able to specify product routes without concern for what resources there are available in the fabrication process, or what other products there may exist in the system simultaneously. A clear separation between the product specifications and the resource models facilitate this. Also, we should be able to model the resources without concern for

what other resources there may exist in a specific application, or what kind of products this process is going to produce. This enhances the flexibility of the system, since we can introduce new resources and new products (within reason) without affecting the modules already present within the control system.

However, for profitable work to be carried out, the modules of the control system must interact. This interaction must be performed in an orderly way to guarantee that all products can be satisfactorily produced. For this, we use the models of the products together with the models of the fabrication resources to synthesize a supervisor that acts as a controller, commanding the resources to carry out tasks in such a way that the resource interactions cannot lead to undesired system states. For control systems of fabrication processes we propose the implementation scheme shown in Figure 5.2. The lion's share of this thesis has been concerned with the supervisor synthesis, but in the rest of this chapter, we will discuss the resource modeling and the compilation of high level product specifications into a format suitable for supervisor synthesis. Resource and product models applicable for control, once the supervisor is generated, are given by Adlemo (1995a), Andréasson (1995) and Gullander (1995). The work presented in this thesis builds on earlier work looking at some of these problems. See page ii. The work has focused on three sub-topics.

**Resource models** – the capabilities, constraints and behavior of the production resources must be adequately modeled. Such a model must also be transformable into a low-level model usable for supervisor synthesis.

**Product models** – the operations necessary to produce a specific product are given as high-level operation lists. These must also be transformable into low-level models, usable for supervisor synthesis, as well as for debugging and manual control.

**Control synthesis** – this includes mapping the operation lists onto the capabilities of the plant for static resource allocation, and to generate a supervisor that dynamically drives the system outside any undesired states, but always able to reach some desired state.

As pointed out by Balemi (1992), two reasons for the supervisory control theory not to have gained a widespread acceptance in industrial applications, are the problem of *model interpretation* and that of *supervisor implementation*. For supervisor synthesis we need models of the plant and the product specifications given as discrete event processes on a high abstraction level, such as state-machines or Petri nets. For implementation of control systems for complex and flexible fabrication processes, there is a need for entirely different kinds of models. An implementor or operator of such a control system would not be content with working with Petri nets or state-machines on a global scale. For product specification, for instance, we would like to be able to merely give the operations that the workpart is to undergo to be satisfactorily produced, together with actions to take at specific points concerning malfunctioning. Then a product can be specified independently of the plant it is to be produced by, and the same specification can be used to produce the same product in different plants. This approach is becoming widely accepted within the chemical batch processing industry, see, for instance, SP-88 and Tittus (1995b). Resource models must also be available in a format suitable for control-system implementation, as

well as for operator interfaces. An operator cannot be presented with a large Petri net representing the global fabrication process.

This inconsistency between what the system should look like to an implementor or operator, and what the system should look like for supervisor synthesis, has to be resolved. One way to do this is to "compile" high-level product specifications and resource models into a format suitable for supervisor synthesis; see Figure 5.2. Such a compilation must be done very carefully, so as to maintain the connection between the two levels of views.

The topic of control synthesis above reveals that we will view the supervisor in the Balemi-framework, the input/output interpretation, where the supervisor generates command events, and the plant generates response events. The response events are considered to be uncontrollable, so that, once the supervisor has sent a command to initiate some action, it must always be able to accept the response. This work has been continued by a cross-departmental research project at Chalmers University of Technology, looking also at operator interfaces and structured design methodologies for flexible fabrication processes. Our present goal is to answer the question: where does the supervisor fit into all of this?

## 5.2 Object Oriented Resource Modeling

The resource models are based on an explicit mapping between elements of the physical system and modules of the control system. Using object-oriented principles, see Shlaer (1992), we create *internal resources* corresponding to the physical devices. For each physical device there exists a corresponding internal resource, handling the communication between that physical device and the other physical devices through their corresponding internal resources. The functionality implementation and I/O-demands of each physical device is encapsulated within a closed module. This readily structures the system in a coherent, logical way. The control system is modularized, and the I/O-requirements of the system are partitioned into logically connected units. Furthermore, the concept of an internal resource brings the possibility to raise the modules to equal levels of "intelligence", even though they may be of varying sophisticacy in the real world. For instance, a collection of a number of low-level devices, such as air-pressure cylinders, can be viewed as a unified device. Thus, as seen from the controller and the other internal resources all devices can be made to look equally intelligent.

For reusability of the internal resources, it is imperative that they are as decoupled as possible. Therefore, the synchronization of the system is administered by a separate entity–the controller. This is the only module that has to know every resource in the system, thus it will contain and encapsulate the system-specific aspects. This separates the control of the individual devices from the control of the whole system. To make the internal resources general they are partitioned into a *general part* comprising general information common to all similar devices, and a *specific part* that particularizes this information, adapting it to the actual physical device. See Figure 5.3.

The general part is described by a DEP, typically an automaton or a Petri net, representing the behavior of the physical device. Thus, the general part keeps track of the current state of the external resource. The internal resources have a client/server relationship, and synchronize their activities using high-level messages. These messages are sent and received by the general part, that executes resource-specific tasks upon receipt of

a message. The specific part has to translate the high-level messages exchanged between the internal resources into some (lower-level) proprietary protocol that can be understood by the physical device. In some cases this translation may simply be to relay the received message to the physical device. In other cases, though, when the functionality promised by the general part is not available by the physical device itself, this functionality is implemented within the specific part. In many cases this leads to the execution of specific state-machines, and we may thus achieve a hierarchical resource model.

Physical devices can be collected into *classes* with similar behavior on an abstract level. All such devices are modeled similarly, by the same DEP, and thus the internal resource provides a homogenous interface to the other internal resources as well as to the supervisor. The same general part would be used for all internal resources modeling physical devices of the same class; only the specific parts would have to be adapted for each specific physical device. In an object-oriented environment the specific part would be implemented as *virtual functions*, see Shlaer (1992), that are filled in at instantiation time. It is not hard to conceive of the specific part being offered by the device vendor, ready to be plugged in when the application is configured. See Cox (1986) for an interesting discussion on such "software IC's".

The internal resources modularizes the control system, and their message interchanging makes them loosely coupled. This has the following benefits.

**Flexibility** – the flexibility of the control system is enhanced, since internal resources can be added, replaced or reprogrammed, without affecting any of the other modules.

**Reusability** – the general parts are not specific to vendor, model, calibration etc., this is all collected in the specific part. Therefore the general part is reusable.

**Understanding** – the structure of the control system adheres closely to the structure of the physical system, thus making it intuitive and easy to understand.

**Feasible design** – since the internal resource provides a high-level interface to the physical device, implementation of the supervisor is simplified. The supervisor only has to consider high-level "intelligent" tasks, the internal resource handles local tasks and communication with other internal resources, so called *handshaking*.

Three classes of resources have been identified, and for these Gullander (1995) has developed Generic Resource Models (GeRMs). The generic resource models are based on the ones presented in [3] and experiences from several case studies as well as from the PAC model developed by the ESPRIT project COSIMA, see Bauer (1991). Note though, that the GeRMs that have been developed by Gullander (1995) are not suitable for assembly systems or batch processes. Constructs very similar to GeRMs applicable to batch processes are presented by Tittus (1995b).

The identified resource classes are[2]:

**Producers** – devices that make physical or logical changes in product properties, such as lathes (physical) or quality control stations (logical).

[2]These were originally called *machines*, *buffers* and *robots* in [3], but the current terminology seems more general.

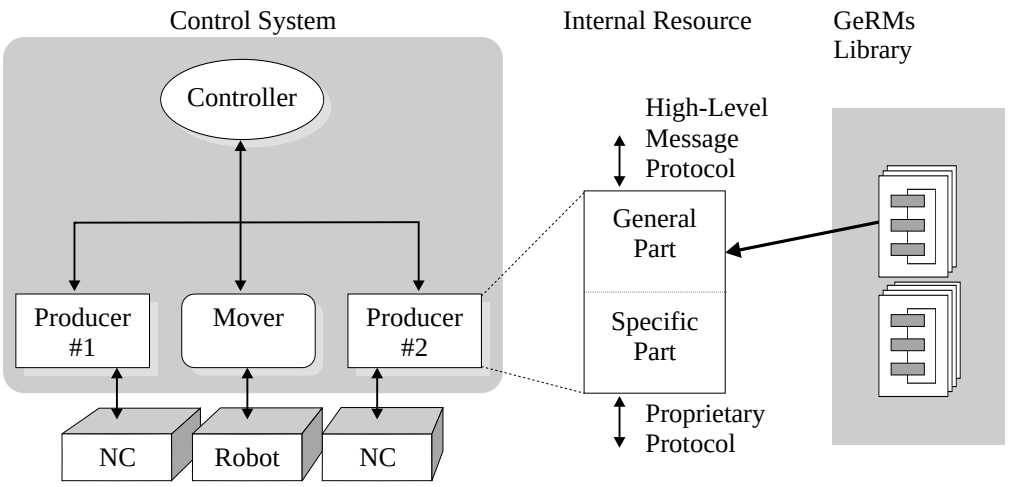



**Figure 5.3:** The proposed control system structure. The physical devices at the bottom left, are the external resources, while their corresponding software modules within the control system are the internal resources.

**Locations** – devices that provide intermediary storage for products, such as buffers.

**Movers** – devices that transport products between producers and/or locations.

In Gullander (1995) a Generic Message Passing Structure (GeMPS) describing the interaction between controller and resources, as well as between resources is presented. The purpose of the GeMPS is to support system development, giving a set of messages and a message-passing structure that together will lead to a modular and flexible system, easy to understand and simple to implement. GeMPS focuses on the interaction between internal resources and the controller necessary for moving parts from one position to another, loading parts, processing and then unloading parts. See Gullander (1995). A GeRMs library, see Figure 5.3, further facilitates the implementation of control systems for flexible fabrication processes.

As can be seen from Example 5.1, there are two types of messages described by GeMPS

**operation messages** – sent between the controller and the resources. These are the *commands* Take, Release, and Process and the *responses* Taken, Released and Processed.

**handshake messages** – exchanged between movers and producers (and locations). These messages are used to perform local tasks, typically loading and unloading of a machine.

For specific details on the application of the proposed GeMPS to manufacturing systems, we refer the reader to Gullander (1995). Example 5.1 describes a modified form of the GeMPS, suitable for the controller implementation that will be described in this work. Refer also to [1], [2] and [3]. To us, the benefit comes from the fact that we can assume that the system is controlled by the commands and responses, only. Locally, the resources perform handshaking when requested by the controller to perform a task. Thus, the

### Example 5.1 Generic Message Passing Structure

Illustration of the modified Generic Message Passing Structure of a control system.

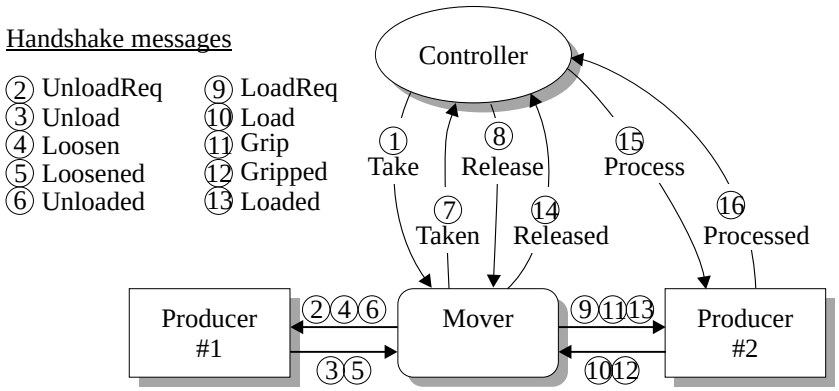



**Figure 5.4:** The generic message structure proposed. The handshake messages are listed to the left, while the messages 1, 7, 8 and 14 to 16 are the command messages. The internal resources are shown at the bottom.

This scenario describes an unload-move-load procedure initiated by the controller. The mover first receives a Take message from the controller, instructing the mover to move to producer #1 to unload it. When the mover arrives it informs the producer by sending an UnloadReq message. The producer replies with an Unload message, when it is ready to have the part unloaded. The mover then moves into position to fetch the workpart, and sends a request to the producer to release the part, Loosen. When the part has been released, the producer replies by sending a Loosened message. Then, the mover exits the producer and informs it that the unload procedure is finished by sending an Unloaded message. The mover also responds to the controller with a Taken message.

By sending a Release message, the controller then instructs the mover to load producer #2 with the product. After moving the part to producer #2, the loading is performed using the LoadReq, Load, Grip, Gripped and Loaded messages, in a similar way as the unloading of producer #1. After completion of the load procedure, the mover informs the controller that the move-load task has been carried out successfully. This is done by the Released message. The controller then orders the producer to start the machining of the part, by sending a Process message. When the processing has finished, the producer sends a Processed message back to the controller.

■

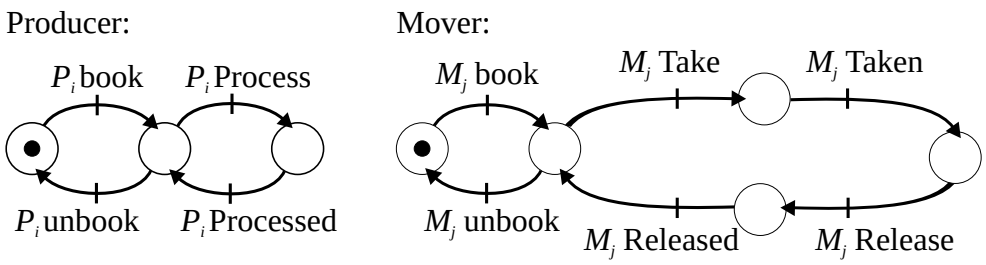



**Figure 5.5:** Generic models of the Producer and the Mover classes, used for supervisor synthesis.

controller chooses what commands to issue and receives responses upon completion of those commands. However, in addition, the controller needs to coordinate the dynamic usage of the resources so as not to issue conflicting commands or commands that take the system to unwanted states; it has to know what actions can be performed when. This coordination is expressed by the supervisor, in allowing or preventing specific products access to specific resources at specific system-states. For the following discussion, we will distinguish between these two aspects of control, commanding and coordinating, by saying that the controller commands and the supervisor coordinates. However, there is really no need to distinguish between the supervisor and the controller, a controller is just an active supervisor that issues commands. See also Balemi (1992).

For dynamic resource coordination we need to model whether a resource is being free for some task, or not. If it is free, that resource may be designated to that task if this does not incur blocking of the system. However, we also need to model the commanding parts of the resources. The resource models we have found adequate for supervisory control are shown in Figure 5.5.

In Figure 5.5 we can note that the Process, Take and Release commands are present, as are the Processed, Taken and Released responses. But there are also two events "book" and "unbook" for each resource, the *synchronization events*. These, as well as the commands and responses are specific for each internal resource, so that all resources have disjoint event-sets as far as the supervisor is concerned. For supervisor synthesis, the responses are considered to be uncontrollable. We can decide when to initiate a task at some resource, but we cannot know when or prevent that task from being finished. The aspect of resource failure is not being modeled, but it is obvious that it can be included in the model. When, how and what types of failures to include, is still an open question.

The book and unbook events keep track of whether a specific resource is available for a task, or not. Naturally they are controllable, and they permit the supervisor to allow or prevent a product access to the resource. Thus, these events constitute the synchronization events, used by the supervisor to determine what commands to allow the controller to generate. When in a system-state where a resource is booked, it is guaranteed to be safe to issue the commands, and any of the commands allowed in a specific state are valid. In this way the supervisor restricts the command sequences of the controller, so as to guarantee that the closed-loop system always stays outside the forbidden states and is nonblocking.

We can note that the models presented in Figure 5.5 are very similar to the ones given

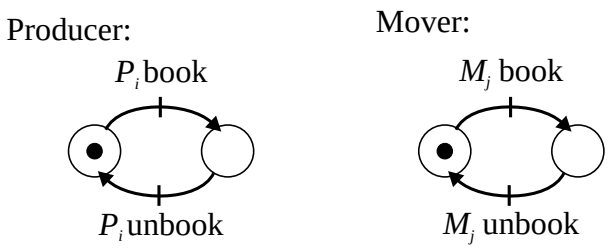



**Figure 5.6:** The synchronization models of the generic resource models. The events are in this thesis denoted by $a_i$ and $c_i$, for a producer **Mi**, and $x_j$, $y_j$ for a mover **Rj**

by Tittus (1995b) for control of chemical batch processes. The producer model seems to be exactly the same, while the mover model should be compared to the *connection-line* of Tittus. For batch processes the movers do not actually move, but they remain fixed and only move the product. Furthermore, the product is a continuum as far as moving it is concerned. Thus, the product occupies both the resource to be moved from and the resource to be moved to, as well as the movers, simultaneously during transportation. This is not so for manufacturing systems, however. In a manufacturing (and assembly) system the product is a discrete component (a single product or a pallet, for instance), that only occupies the mover during transportation. It is mainly this that makes the mover models so different for manufacturing systems and batch processes. The concepts are the same though.

For the purpose of synthesizing a supervisor coordinating the sharing of the resources among the simultaneously running product routes, it often suffices to model the resources even simpler than those shown in Figure 5.5. In the applications we have been looking at, it seems enough to model the resources as two-state processes; the states essentially meaning IDLE and BUSY, or BOOKED and UNBOOKED. See Figure 5.6. This is also recognized by Zhou (1993), where such states, or rather places, are called *B-places*. See also the work of Banaszak (1990). When a resource is in its BOOKED state it is valid to send the operation commands.

The models of Figure 5.6 will be referred to as *synchronization models*, see Tittus (1995c), and we will use these in the following. See also Figure 1.3. In more complex settings, the models may have to be more complicated, but it seems that that two states conveying the information that the resource is free or occupied are always present. See, for instance, Tittus (1995c), where the resources are modeled, essentially, by three states, BOOKED, UNBOOKED and PROCESSING. In the third state a continuos (or hybrid) controller local to the device executes to take the batch from some continuos state to another.

For systems with multiple movers that can get in the way of each other, the mover routes have to be modeled also, so that the supervisor can coordinate the movers in their actions. Such systems, though covered by the theory of the previous chapters, are not considered in this chapter, other than when two movers can collide if the simultaneously try to load and unload the same producer. Furthermore, for error recovery some states and events representing failures have to be included but the IDLE and BUSY states are still present. A resource can only break-down when it is performing some work, that is, when it is in the BUSY state. In this thesis, we do consider resource failures.

## 5.3 Product Descriptions

The use of high-level product descriptions, *operation lists* by Andréasson (1995) and *general recipes* by Tittus (1995d), is motivated by a desire for increasingly flexible fabrication processes. Flexibility inherently demands a separation between the manufacturing resources, the product routes and the control system. Such a separation permits incorporation of new or different equipment as well as new products without re-programming the control system or re-configuring the manufacturing resources. Thus, the system can automatically survive changes in its environment, meaning that the system is reusable between applications, since it can be used in different environments.

Operation lists can come in different forms, and on different abstraction levels, each suitable for its own use. The high-level operation lists permits people with a moderate technical background and limited knowledge of the system to profitably interact with it on a high abstraction level. However, this high abstraction level demands some kind of support for managing the connection between the specified product routes and the available machining capabilities. There has to exist software tools for handling the high-level operation lists, as well as automatically transforming them into lower level forms usable for the control system. For instance, supervisor synthesis requires Petri net (or automata) descriptions of the actual product routes.

High level operation lists can be given in either algebraic or graphical form. In this section a suggestion for a graphical form, together with its Petri net transformation will be described, while an algebraic approach is described in Andréasson (1995).

Others have found equal or similar description mechanisms useful for specifying and modeling production steps and work cell operations in a highly structured way. Compare, for instance, the Process Activity Language (PAL), see Krieger (1993). Several of the operators present in PAL will also be specified in this section, though in some cases renamed. Other operators defined in this text are not present in PAL, and yet others have slightly different semantics. PAL is focused on simulating and verifying product specifications, while our work has as a goal to facilitate synthesis of correct control laws for the production of the specified products. Also, unlike PAL which bases its graphical representation on ladder diagram, we base our graphical representation on Petri nets, though on a higher abstraction level. All in all, though, it seems that there are at least two research communities that have independently come to very alike conclusions concerning the functionality that has to be present in order to express the operations of a flexible fabrication process in a structured high-level implementation-independent way.

### 5.3.1 High-Level Operation Lists

The preferred way to specify operation lists is graphical. The operator lays out the desired product routes, focusing merely on what operations to perform and in which order. A graphical layout is well suited for a computerized tool with a graphical interface. For combining operations a number of general constructs will be described with examples of useful applications.

A set of high-level operators is a fine tool for specifying operation lists. However, for supervisor synthesis and for verification and simulation of the specifications, we have to transform the high-level operators into a mathematically more strict, lower level formal-

ism. For this we will use bounded Petri nets, Peterson (1981), because of their well-known and well-defined mathematical properties. It is also well-known that a bounded Petri net can always be equivalently expressed as a finite state automaton.

The key to Petri net transformation of the described high-level operators is the association of every operation with an event symbolizing that operation. Each such event labels a transition in the Petri net, and has a corresponding event, equally labelled, in (at least) one resource. The resources may also include other events, not necessarily symbolizing operations of interest to the operation list, such as PartLoaded, MachineReset, etc. However, only operations defined by the manufacturing resources can be used in specifying the operation list.

In the Petri net formalism, the firing (execution) of an event is considered to encompass zero time. Thus, the modeled events are considered to be *primitive events*, see Peterson (1981). Primitive events are instantaneous and nonsimultaneous. Operations in the real world, however, take time and are thus nonprimitive events. Nonprimitive events are not nonsimultaneous, and hence may overlap in time. Nonprimitive events cannot properly be modeled by single transitions in Petri nets, but have to be decomposed into two primitive events, "nonprimitive event starts" and "nonprimitive event finishes", together with an intermediate place representing "nonprimitive event occurring". Therefore, for an operation (nonprimitive event), $Op$, we introduce two transitions into the Petri net, labeled $Op$ and $\overline{Op}$, representing the primitive events. We will call $\overline{Op}$ the complement of $Op$, compare Milner (1989) and Section 2.5.

Note that, "operation" is informally defined here. An operation can exist of sequences of actions, which could themselves be regarded as operations. This means that the places representing, "nonprimitive event (operation) occurring", can be regarded as "macro steps" representing arbitrary sequences of the defined constructs. Note also that an "operation" is not necessarily an action of physical nature, but may also represent "abstract" actions, such as the claiming of a resource for exclusive use, that is, *booking*, see Tittus (1995a). In fact, this is how we will use the Petri net transformations, see Section 5.4.

### Sequence and Alternative

In Figure 5.7 can be seen the graphical representation of the Sequence and Alternative constructs. As can be seen an operation is graphically represented by a rounded box with the name of the operation. The type of product for which the route is specified is also named, and represented by a circle. Arrows between the operations define their sequential order. Note also that, contrary to standard precedence graphs, multiple outgoing arcs from an operation (or product) represent a choice of one, and only one, arc to follow. Compare to the dynamic precedence graph of Valckenaers (1994). Sequence actions typically represent constraints on the product given either by the physical layout of the manufacturing cell, or the product itself. For instance, before a hole can be tapped it has to be drilled. The Alternative ("case" in the vocabulary of Krieger (1993)) construct is more subtle. An example might be that there are several machines performing the same operations by different means, such as metal cutting by laser, plasma or water jets. For some products these operations must be distinguished due to material considerations, whereas other products can be handled by any of these machines. Being able to express a

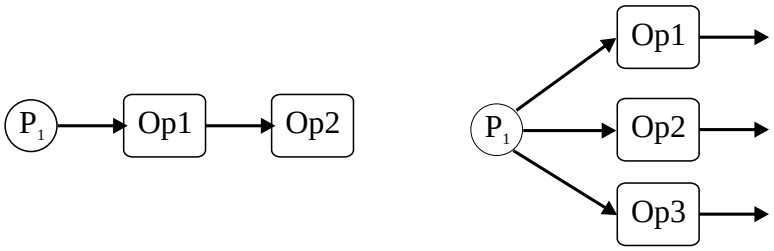



**Figure 5.7:** Graphical high-level representation of the Sequence (left) and Alternative (right) constructs.

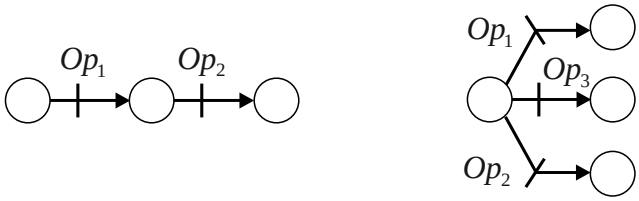



**Figure 5.8:** The Petri net transformations of the Sequence (left) and the Alternative (right) constructs. Notice the similarity between the Petri nets and the high-level constructs of Figure 5.7.

selection for these products would thus increase throughput, since the currently available cutting machine can be used.

The Alternative construct is thus used to choose a specific operation between a number of allowed operations. The control system is allowed to choose any of these, presumably the first to be available. However, should several operations be available simultaneously, the choice is nondeterministic. In practice, some choice has to be made by the control system, but this is a run time implementation issue. In some cases the nondeterministic choice may be desirable, but in other cases it may not. To be able to exercise more control over the actual run time choice made between several simultaneously available operations, there can be set priorities between the alternatives. The uppermost (or leftmost) branch can, for instance be considered to have the highest priority and thus the bottom(rightmost) branch the lowest. More detailed priority structures can be defined by assigning a priority number to each alternative.

The Sequence and Alternative constructs are trivially transformed into Petri nets, see Figure 5.8. Transforming the Sequence construct is merely a matter of labeling transitions in a sequence within the net. The places in-between would then symbolize occupying a resource, being operated upon and (possibly) being transported to another resource for the next operation. Note that in the case of the Sequence construct, the comlpement event does not seem to be necessary. The reason for this is that the strict sequence does not include the possibility to start the next operation before the current one has finished. This is inherent in the sequence construct.

Of course, more elaborate actions can also be taken in sequence or in alternative cases; sequences can consist of sequences and alternatives, while alternatives may contain other

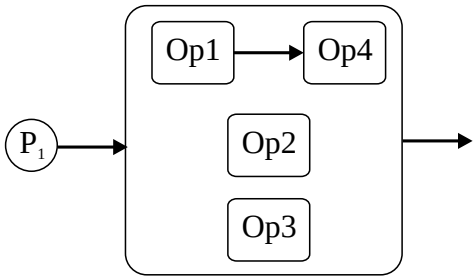



**Figure 5.9:** The Unconstrained Order construct. The operations are allowed to be carried out in any order, but *Op1* and *Op4* cannot be interleaved.

alternatives in sequence.

### Unconstrained Order

The *Unconstrained Order* construct is actually a special combination of the Sequence and Alternative constructs and could thus be expressed by these. However, for operation list specification we have found the Unconstrained Order construct very useful. Many operations do not incur any constraints on their order as far as the product is concerned; a product may in large be operated upon by the manufacturing devices in any order. For instance, regarding automobile rear axles, it is clear that a milling operation on the rear axle's 'banjo', and a lathe operation on the rear axle ends can be done in any order. The real constraints come from the physical layout of the system and the available transport routes.

The Unconstrained Order construct has to be adequately distinguished from the Alternative construct with a clear, logical and intuitive graphical representation. We have chosen to collect the operations of the Unconstrained Order construct within a bounding box, as is seen in . This seems a logical representation in that all the operations within the box are to be carried out before execution can continue outside the box. Absence of arrows within the box automatically describe unconstrained order, whereas arrows present within the box determine constraints. This can be seen in Figure 5.9 where *Op1* and *Op4* are to be carried out in sequence, but no constraints are specified between *Op1/Op4*, *Op2* and *Op3*. Note, that (though other interpretations are possible) in Figure 5.9 *Op2* or *Op3* *cannot* interleave *Op1* and *Op4*. The bounding box can easily be specified by means of a pointing device (mouse) after other constraints have been introduced (though this is no requisite). Thus, inside the box, any of the available constructs can be used.

Trivially, the Unconstrained Order construct can be transformed into a large Petri net describing all the possible operation routes, see Figure 5.10, left. This net is really the transformation of the Unconstrained Order construct expressed in the Sequence and Alternative constructs. However, there exists a more sophisticated way to express the Unconstrained Order construct in Petri net form. In Figure 5.10 to the right, is shown a net with substantially fewer places that adequately describes the construct. The top transition is the "exit" transition of the former product specification construct, while the bottom transition is the "entry" transition of the next product specification construct.

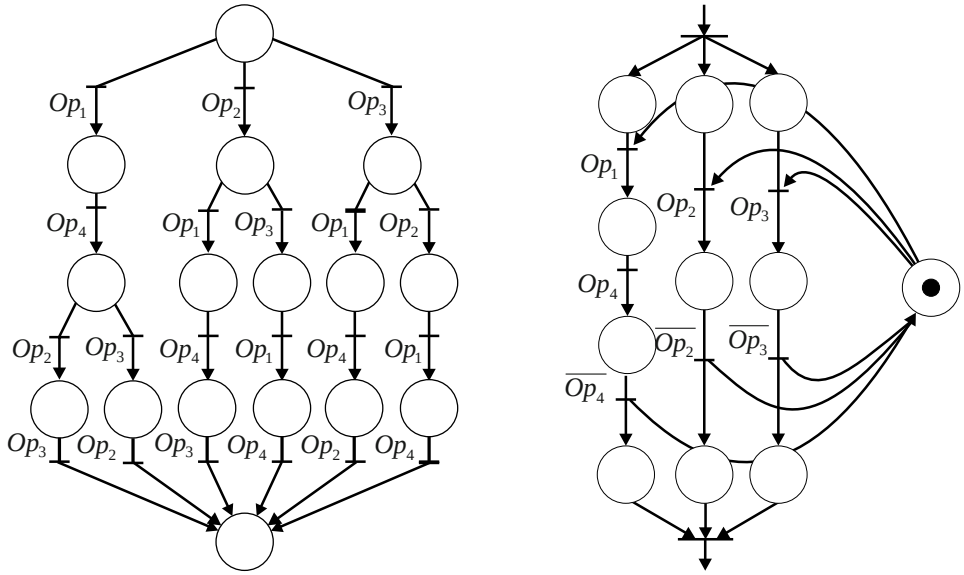



**Figure 5.10:** The trivial Petri net respresentation (left) and the more sophisticated representation (right). In comparison with Figure 5.9, the benefits of a high-level representation is effectively illustrated.

Note that, though the two nets in Figure 5.10 are equal from a sequencing point of view, they are different in the interpretation of the transitions and places. As mentioned above, in the net of Figure 5.10, left, a place represents occupying a resource, being operated upon and being transported to another resource for the next operation, the net of Figure 5.10, right, also includes a transition representing "finished operating upon". This is the transition that returns the token to the single mutual exclusion place to the right. This is imperative for a correct interpretation of the net; no other operation can (physically) begin before the current one has been finished. Thus, mutual exclusion of the operations is inherent in the sequences of Figure 5.10, left, while it must be explicitly introduced into the net of Figure 5.10, right. Observe though, that the complement event for *Op1* is not included in the net. Naturally, this a consequence of *Op1* being in sequence with *Op4*. Since no interleaving of the sequence is allowed, any other operation cannot start before *Op4* has finished.

To introduce the possibility of parallel actions, the net of Figure 5.10, right, could be constructed with multiple tokens in the mutual exclusion place. With only one token in that place, the Unconstrained Order will not allow parallel actions, with multiple tokens it will. Observe though, that in this way we cannot explicitly specify a subset of operations that may be performed simultaneously. Either any specified number of operations can all occur simultaneously, or none. Furthermore, with multiple tokens, interleaving of the *Op1/Op4* sequence with the other operations cannot be prohibited.

When is the Unconstrained Order construct useful? As mentioned above, most constraints on order of operation is given by the manufacturing devices and the physical layout of the system. Thus, the Unconstrained Order is a useful means to incorporate as weak constraints as possible into the specification of the operation order. This facilitates

a flexible overall system, in that altering the physical system does not incur altering the operation lists. For example, in one application, including a milling machine and a lathe, an overcapacity in the mill was discovered. To increase throughput a second lathe was planned to be introduced into the system, and both lathes were to take over some work from the mill. The new physical system thus obtained would mean re-coding of the operation lists, had they been specified with the constraints of what operation to be performed where and in what order. Using the Unconstrained Order construct would unnecessitate any change of the operation lists, whatever physical layout of the new system arised; placing the new lathe before or after the mill, for example.

### Synchronous and Asynchronous Divergence and Convergence

The difference between the Synchronous and Asynchronous Divergence constructs is delicate. Both involve simultaneous actions. However, the Asynchronous Divergence construct expresses operations that may be executed concurrently, whereas the Synchronous Divergence construct requires synchronous activation of the operations. Thus, there is a difference between synchronous and non-synchronous activation of the involved operations. In both cases, however, all operations must have been performed before the construct is considered to be completed. The synchronous and asynchronous constructs are distinguished by a filled and non-filled transition, as seen in Figure 5.11. The filled transition can thus be seen as representing a harder constraint on the operations than the unfilled transition.

Convergence constructs denote ways to merge two (or more) products into one. The operation lists of the hitherto independent products should thus meet in an assembly operation. To mirror the divergence constructs described above, two useful convergence operators have been defined; Synchronous and Asynchronous Convergence. These are also shown in Figure 5.11. With the Synchronous Convergence construct the operation of assembly cannot start until both "operands" (products, for instance) are present. The Asynchronous Convergence construct however, allows the operation to start with just one operand. Note though that both assembly operations cannot finish until both operands have been fully processed.

The Synchronous Convergence construct represents the "normal" assembly operation within manufacturing processes. The assemblage of two products cannot begin until both products have been successfully loaded and possibly preprocessed. The Asynchronous Convergence construct, on the other hand, could represent operations within (chemical) processing industries. For example, in a mixing operation where one tank is filled from two separate tanks, the filling must not (always) be synchronous. One tank could start emptying its contents into the mixer tank, while the other tank were still heating its contents. Of course, the mixing operation would not be finished until the contents of both tanks had been mixed (and stirred) in the mixer tank. Note that if both tanks were simultaneously ready to empty their contents, it would be nondeterministically decided which one would begin.

In Tittus (1995b) is defined the General Join, consisting of a (possibly empty) presynch phase, a synchronization point and concluding with a (possibly empty) postsynch phase. In the presynch phase both material flows start independently of each other, and execute until the synchronization point is reached. In the postsynch phase the two, hitherto

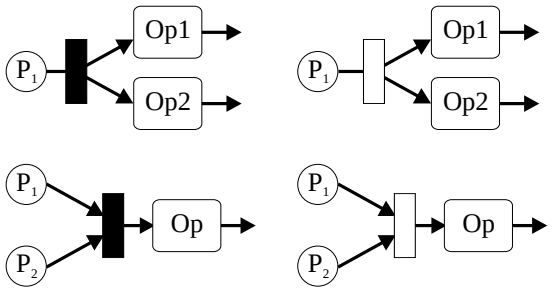



**Figure 5.11:** High level, graphical representation of the Synchronous (left) and Asynchronous (right) Divergence (top) and Convergence (bottom) constructs.

self-sustained, flows are no longer independent. The Synchronous and Asynchronous Convergence constructs are equal to the extreme cases of the General Join. The Synchronous Convergence corresponds to the Synchronous Join of Tittus (1995b), where the presynch phase is nonexistent. The Asynchronous Convergence, on the other hand, corresponds to the Asynchronous Join which lacks the postsynch phase. Note though that in Tittus (1995b) is included the possibility to specify how the convergence should behave; a specification that is required to be upheld during the convergence. For instance, during convergence of two (chemical) batches, there may exist constraints on the maximal deviation from the specified relative amounts of substance per time unit.

When considering the Petri net forms of the Synchronous and Asynchronous Divergence and Convergence constructs, it is beneficial to first consider an abstract device performing the divergence or convergence. Divergence represents the splitting of one product into two products, sawing or cutting, for instance, while convergence represents assembly, welding, gluing, riveting or, as for batch processes, mixing. Neither of these actions are spontaneous, the product cannot cut itself in two. There has to occur some physical operation for the divergence and convergence to take place. In the following we will discuss convergence to some depth. Divergence is the dual of convergence, so much that the Petri net describing the action of divergence is equal to the Petri net describing convergence. Only the initial marking has to be changed, as will be shown. Thus, for divergence an equivalent discussion as for convergence is applicable.

Convergence is a matter of loading two distinct products into a resource, which then performs an assembly operation, whereafter one product is emitted. The actual convergence operation must, of course, be synchronous with respect to both products. However, for such an Abstract Convergence Machine to be general, some additional aspects have to be considered. First, since two separate products are loaded, this can occur in any order, product **A** can be loaded before product **B**, or vice versa. But, there may also exist convergence resources that can load both products simultaneously, an assembly unit with two loading docks, for instance. This is especially important for mixing operations in batch applications where the simultaneous filling (loading) of a tank (resource) with both components (products) may be required to keep their relative amounts constant. Second, preprocessing may be applied to one or both of the products, before they are assembled. This preprocessing will be regarded as part of the loading though. In the same way will we regard any postprocessing, deburring, washing, etc., as part of the assembly operation.

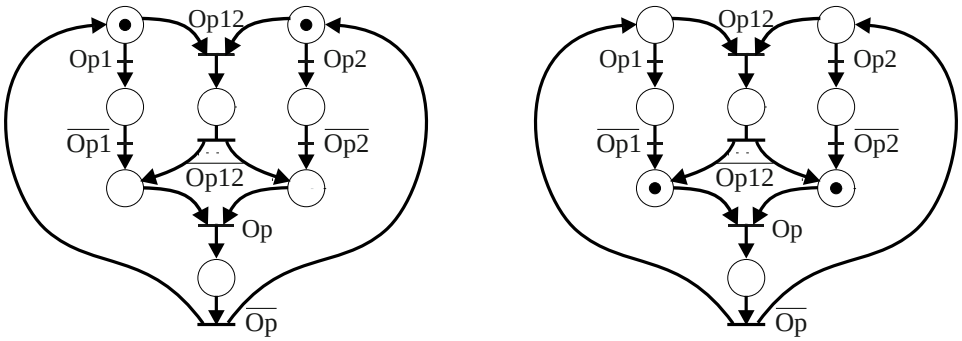



**Figure 5.12:** The Abstract Convergence (right) and Divergence (left) Machine. *Op1*, *Op2* and *Op12* represent loading/unloading and pre/post processing of the **A**, **B** and simultaneously the **A** and **B** products, respectively. *Op* represents the assembly/disassembly operation.

Note that, for divergence the opposite holds; one product is loaded and so preprocessing can only be applied to this product, while two products are emitted, thus postprocessing can be applied for either. In this case, preprocessing is part of the divergence operation, while postprocessing is part of the unloading operation.

Thus, the Abstract Convergence Machine must include, not only the possibility of loading each product separately, but also loading both concurrently. In the Petri net formalism, this may be implemented as three separately labeled transitions; one representing (the start of) loading and preprocessing of product **A**, one representing the same for **B**, and a third transition representing the simultaneous loading of **A** and **B**. The Abstract Convergence and Divergence Machines are shown in Figure 5.12. We can note that the Abstract Divergence Machine is the dual to the Abstract Convergence Machine. One is obtained from the other, simply by altering the initial marking, see Figure 5.12. The Abstract Divergence Machine starts with the loading and disassembly operation, and then the parts can be postprocessed and unloaded separately or simultaneously. The Abstract Convergence Machine starts with the loading, either synchronously or asynchronously, and then assembles the parts and emits a finished product..

An alternative to specifying the third event, representing synchronous loading, is given in Tittus (1995d), where the Petri net formalism is extended to include event connection. This is a way of specifying synchronous occurrences of different events. Thus, the notation means that transitions labeled with a and b, respectively, should fire synchronously. This extension of the Petri net formalism is motivated by a need to model mixing operations in chemical batch processes. Two products, in liquid form, can then demand synchronous mixing, as opposed to first pouring one of them into a tank, and then the other.

With the Abstract Convergence Machine, it is quite clear how the high-level operation lists should be interpreted as Petri nets. The Synchronous and Asynchronous Convergence constructs are shown in Figure 5.13. Note that the high-level operation list only has to specify the (start of the) loading operations. For the divergence construct, simply reverse all arcs and replace all operations with their complements.

The Synchronous Convergence construct of Figure 5.13, synchronized with the Abstract Convergence Machine of Figure 5.12, is shown in Figure 5.14. Note that the places

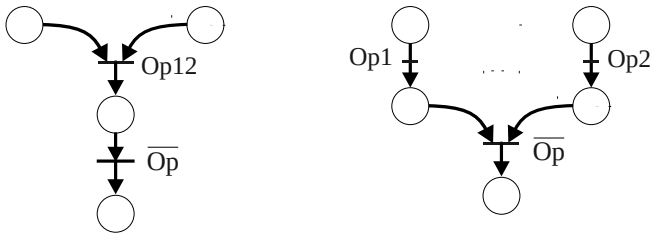



**Figure 5.13:** The Synchronous (left) and Asynchronous (right) Convergence constructs.

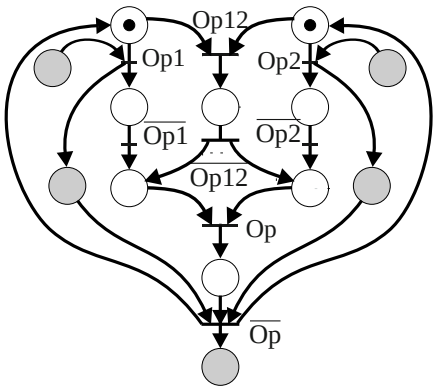



**Figure 5.14:** The full synchronous composition of the Abstract Convergence Machine and an operation list describing Asynchronous Convergence.

corresponding to the operation list are grey in Figure 5.14. Since the Abstract Convergence Machine can load the products either synchronously or asynchronously, it is the operation list that determines which action to take. With the Asynchronous Convergence construct, the action of loading and preprocessing one product is allowed to start, even though the other product may not yet be available. However, at some point both products must be present for the operation of assembly to begin. With the Synchronous Convergence construct neither the operation of loading and preprocessing, nor the operation of assembling can begin unless both products are present.

## 5.4 From Operation List to Final Specification

As described above, the operation lists, can be transformed into Petri nets describing the sequence of operations necessary for the fabrication of the product. In Tittus (1995a) such a Petri net is used as a *mapping recipe* that can be mapped onto a model of the connectivity of the plant, the *mapping model*. The mapping recipe is not specific to the plant that is going to form the product. Only the relevant operations and their relations are specified. Naturally, for the product specifications to be usable for supervisor synthesis, specific resources of the specific plant have to be singled out; *static resource allocation* have to made. This is made for one product at a time, with no concern for any other products that may be present in the system simultaneously. The result of the static resource allocation

is a high-level product route, a *master recipe* in the terminology of Tittus (1995a). The high-level product route can also be translated into a Petri net, in a manner similar to the operation list. In fact, the same Petri net constructs will be used for the transformation of the high-level product route into its Petri net representation. The model of the plant used for this is the same as the model that will be used for supervisor synthesis, the synchronization models of Figure 5.6. The generated Petri net will, however, include *connected events*, pairs of events representing sequences that cannot be interleaved by other events. These event-pairs represent the passing of a product from one resource to another, typically between a mover and a producer.

Once we have the product routes, we have to compose these into a joint specification on the plant. In Section 2.5.1 was given good reasons for model the sharing of the resources among the product routes by *interleaving*. Thus, we will interleave the product routes to form the *local specification*. This specification is termed *local*, since it does not necessarily include the entire alphabet of the plant. To be strict, its alphabet is not even a subset of the plant alphabet, due to the connected events. However, it will be shown that the connected events can be *unfolded* so that the alphabet of the specification is a subset of the plant alphabet. The interleaved product routes contain many interleavings that will not be physically possible. To remove these, and to make the alphabet of the specification equal to the plant alphabet, we synchronize the local specification with the synchronization model of the plant. This synchronization is, strictly speaking, not the same composition as described in Definition 2.43, again, due to the connected events. Once more, though, since the connected events can be unfolded this is in practice not a problem. In the global specification the connected events will finally be unfolded, and auxiliary, product unspecific specifications can be introduced to generate the final specification. This is then used for synthesis of the supervisor.

In this section we will give a very brief example illustrating the generation of a high-level product route from an operation list. We will not describe the necessary models in any detail, merely give the operation list, the physical plant and the resulting high-level product route. For details, we refer the reader to Tittus (1995a) and Tittus (1995c). Then we describe the generation of the local specification together with reasons for introducing connected events.

### 5.4.1 Resource Allocation

The resource models needed for the static resource allocation are different from the ones needed for control. When attempting to map the functional requirements of a product specification onto the capabilities and constraints of the resources of the plant, the internal resources of Section 5.2 are not adequate. It is this that leads Tittus (1995c) to speak of "multi-aspect" modeling. Different design activities, like the resource allocation described in this section, like supervisor synthesis and product specification, concern different aspects of the plant and so need different representations of the resources.

For the purposes of the static resource allocation, *mapping* for short, we need to know the functional capabilities and constraints of the resources, and the connectivity between them. In a batch process the connectivity is governed by the physical connections between the resources. Pipes connect tanks and reactors with other tanks and reactors, via valves and pumps, etc. In a manufacturing system, the connectivity is governed by

the available mover routes. If there exists a mover that can transport a part between two resources, then these resources are connected. We have already argued that for a truly flexible system complete connectivity is essential. We can never know beforehand which routes may be specified by products unconceived of at the time of control system implementation. Of course, if the system possesses full connectivity, then there is no need to model this explicitly. However, when full connectivity is not available, then the connectivity is modeled by the mover resources.

The functional capabilities of the producers include the available operations that the producer can perform. These operations are modeled as parametrized events, with unique standardized names. These names are standardized so as to allow a mapping with the corresponding functional requirements of the product specification. The number of parameters is predefined, and they indicate the physical or safety constraints of the operation. An *operation* is in this sense a task, such as DRILLING, together with constraints, such as, *maximum 100 mm deep, 30 mm diameter*. For batch processes an operation is typically HEATING, *maximum 100 degrees Celsius*. The connectivity is of course a typical functional capability of the movers.

The plant on which to map the product specification is thus composed from the functional capabilities of the resources. This includes the connectivity, when necessary, and the operations. The task is now to map the product specification onto this plant to generate all routes through the plant that lead to the correct production of the product. Of course, for a large plant with flexible devices, the possible number of such routes can be enormous. Some strategy may be taken that does not generate all routes, but only the fastest one, say. The problem is that the mapping is done on a single product basis, but production in a flexible fabrication process is a multiple product task. Thus, choosing only one "best" way, or maybe even a few "good" ones, does not necessarily mean that when running several different products together through the plant, the overall "best" way can be achieved. For that matter, there is no guarantee at all that, by choosing only one route through the system for each individual product, there can be produced several products simultaneously. In the worst case, the production will also have to be on a single product basis, which is not a good thing. However, *if* we generate all possible routes through the system for each product, *then* we know that the "best" route for all products simultaneously will not be overlooked.

Mapping the product specification onto the plant can be done by associating each operation of the specification with a class of operations of the plant. The basis for the association being that the requirements are satisfied. Thus, a product operation that says DRILLING, *8 mm deep, 4 mm diameter*, can be mapped onto the two different plant operations, DRILLING, *max 50 mm deep, max 10 mm diameter* and DRILLING, *max 100 mm, max 30 mm diameter*, which may be offered by two different resources. In Tittus (1995a) is given an algorithm to perform this. The result is a high-level product route with the possible sequences of producers, locations and movers that will fabricate the product in this specific plant. See Example 5.2.

### 5.4.2 Event Connection

When generating the Petri net representation of the high-level product routes, the synchronization models of the resources are used, in the form given in Figure 5.6. However,

**Example 5.2 From Operation List to Product Route**

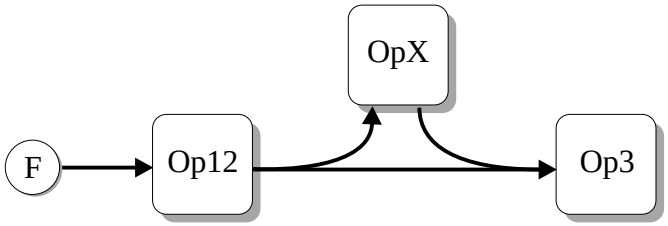



**Figure 5.15:** Operation list for the F-product of Section 6.3.

A part of the operation list for the F-product of Section 6.3 is shown in Figure 5.15. This operation list specifies that the product is to first undergo operation *Op12*, and then operation *Op3*. In-between these two operations the product is allowed to, but not required to, be subject to *OpX*. This means that the product does not require immediate transport between the resources performing *Op12* and *Op3*. This also has the consequence that if a resource existed that could perform both operations *Op12* and *Op3*, but not *OpX* then these operations could be performed in that same resource. In fact, the product does not require the operation *OpX* at all. In this example *OpX* just represents intermediate storage.

The physical plant is given in Figure 6.5 on page 171. In this system there are several movers that transport between the resources. Mover **R1** transports incoming products from the entry stations to either **M1**, **M2** or **M4**. Likewise, the mover **R3** transports products from **M1** or **M2** to the buffer **B1**, and from **B1** to **M3**. The other resources need not concern us at this moment. Since there are multiple movers that do not have full connectivity, the movers must be included in the high-level product routes. This was not the case in the example system of Chapter 1. There we had just one mover able to transport products between every producer, and so the mover was not included in the high-level product route.

Example continued on next page

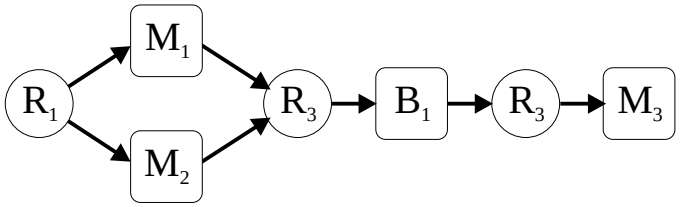



**Figure 5.16:** The high-level product route generated from the operation list of Figure 5.15 for the plant of Figure 6.5

Assuming that the operation *Op12* can be performed by both **M1** and **M2**, the mapping procedure will map *Op12* to both these resources, generating alternative paths in the product route. Transport of incoming products to either **M1** or **M2** is achieved by **R1**, so that the product route begins with this mover. Likewise, transport from **M1** and **M2** is done by **R3**, so that the next step in the product route concerns **R3**. The operation **Op3** can be performed by **M3** only. However, due to physical constraints there is no direct connectivity between **M1** or **M2** and **M3**. Every product unloaded from **M1** or **M2** must be stored in **B1**, before taken to **M3**. It may be that the mover has to change its grip on the product, or that the product has to cool down before being loaded into **M3**. Therefore, the product route will not include a direct path between **M1** or **M2** and **M3**, even though the operation list allows this. Constraints given by the physical structure of this specific plant thus generates a product route that is more limited, in this sense, than the operation list. Note that for some other plant this constraint may not be present and the alternative paths could have been included. The generated high-level product route is shown in Figure 5.16. Its corresponding Petri net product route is given in Figure 5.17. Note that this only describes a part of the product route for the F-product of Section 6.3, as indicated by the broken line to the left in Figure 5.17. Note also that the transitions have been labeled with connected events of the respective resources, except for $x_1$.

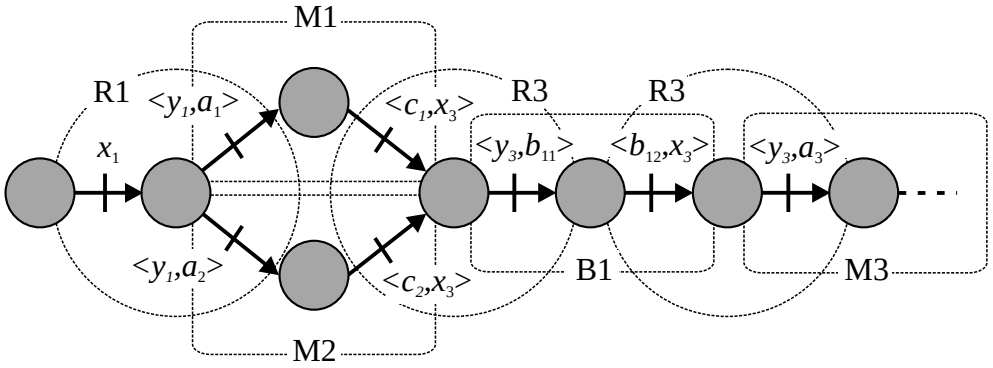



**Figure 5.17:** The Petri net description of the high-level product route shown in Figure 5.16. The respective resources have been shown for reference.

**Example 5.3 Event Connection**

The two problems of not introducing connected events is best illustrated in a state-machine representation.

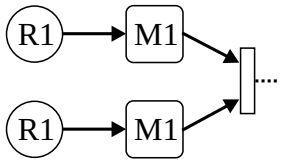



**Figure 5.18:** A partial high-level product route to illustrate event connection.

Assume we have the product route in Petri net form as shown in Figure 5.18. The corresponding synchronization models of the resources can be taken as those shown in Figure 1.3 on page 5. Part of the state-machine representation of the resulting product route is shown in Figure 5.19. Note that this is only a small part. For clarity not all transitions are shown.

Example continued on next page

it does not suffice to generate a Petri net with transitions simply labeled by the synchronization events, since such a net would not adequately describe reality. This is illustrated in example Example 5.3. The problem is twofold. For one, nets without connected event include places in which the product has been unloaded from a producer, say, and not yet entered any other resource. These places arise between the events that we maintain should be regarded as connected. In such a place, the product is non-existent in the system as far as concerns the resources. The product does not occupy any resource, and so all resources are available. Naturally, this is not an adequate model of the system's present state, since the product in fact occupies *two* resources, typically a mover and a producer or location. These resources are the resource immediately preceding that place, and the resource immediately following that event. The second problem concerns the fact that without connected events, one product could be unloaded from a resource and then immediately loaded *as another product*. See the example.

When generating the Petri net representation of the high-level product route, event connection is included by simply joining the events representing unbooking of the previous resource, with the event representing booking of the next resource. Event connection is a consequence of the modeling approach including reusable resource models with product routes tying these resources together. Modeling the system specifically, with no concern for reusability, these states would never be introduced since the exit event of the first resource would be labeled the same as the entry event of the next resource. For instance, Zhou (1993) states that "in general, the stop transition for one activity will be the same as the start transition for the next activity" (Section 2.2.1, page 20). The start and stop transitions of the activities can be equated to the entry and exit events of the respective resources.

Not including connected events would make the resource-models dependent on the product routes. The passing of a product from one resource to another would be modeled

**Example 5.3 continued**

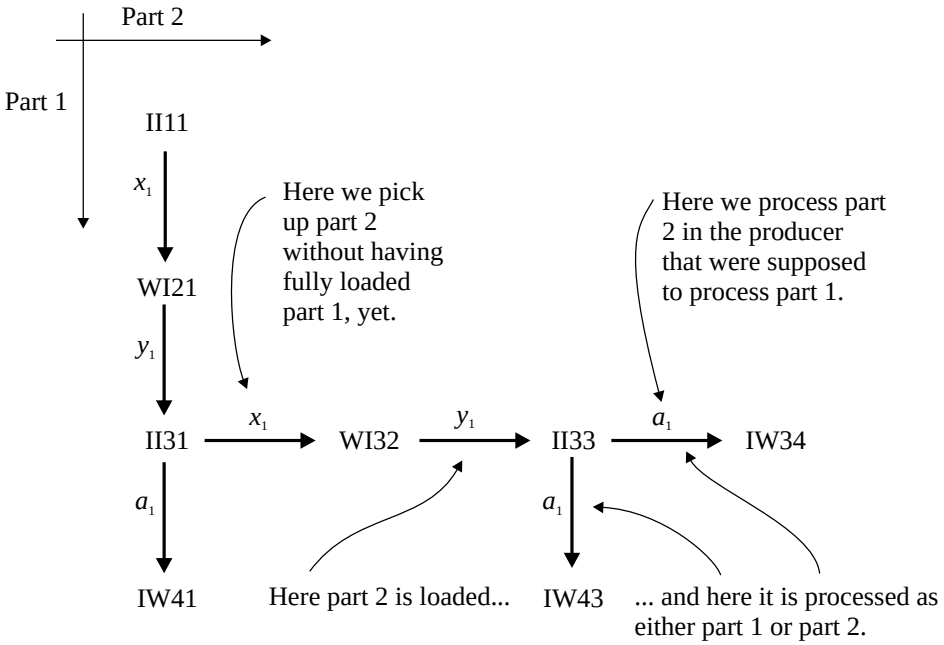



**Figure 5.19:** Part of the state-machine representation of the product route of Figure 5.18 without connected events. The states are named as to represent the states of **R1**, **M1**, part 1 and part 2, in that order. *I* means idle, *W* working.

The two upper explanatory remarks of Figure 5.19 concern the problem of using plant resources that are actually held by part 1, in this case, for the production of part 2, while part 1 is in the intermediary state representing the passing of part 1 between mover 1 and producer 1. In the state II31, part 1 really occupies both resources, but allowing transitions between the $y_1$ and the $a_1$ events, disregards this. Thus, this is not an adequate model of reality. With the connected events $\langle y_1, a_1 \rangle$ we are guaranteed that from the II31 state, $x_1$ cannot occur.

The two lower remarks of Figure 5.19 concern the problem of loading a part as part 2, in this case, and processing it as either a part 1 or a part 2. In state II33 the supervisor cannot determine which of the two paths to take; the horizontal as a part 2, or the vertical as a part 1. (Not unless we assume that there are more information available.) Havoc may reign if a part 2 is suddenly being processed as a part 1. Again, event connection makes sure that the vertical $a_1$ transition does not occur.

In this example the second problem is immediately solved by solving the first one. This is not so in general, though.

■

by a single event in the product route, and both the resources would have to include that event. Thus, each resource would have to include events specific for each product that is to use the resource, and these events would also have exist in the resources that the product is passed to. This is well illustrated by the approach of Banaszak (1990), where the resources include parallel transitions for each product. Connected events avoids this.

### 5.4.3 Local and Global Specifications

The result of the static resource allocation is, for each operation list, a high-level product route describing all possible routes through the plant for this product. This is called the *master recipe* by Tittus (1995a). In the high-level product route resources are appointed to specific operations. To generate the Petri net representation of the high-level product route, it is compared to the plant and a Petri net with connected events is generated. A product route describes the desired paths through the system for *one product only*. Naturally, we want to be able to produce as many products as possible simultaneously. However, since the products are to run as independently as possible through the system, we must have some entity supervising them so as to achieve the goal of having all products produced. For us, this entity is of course, a supervisor. It would be nice if we could generate a supervisor by looking at each product route separately. Then we would not have to deal with the exponential complexity of regarding all product routes simultaneously. Unfortunately, such a scheme does not necessarily generate a minimally restrictive and nonblocking supervisor.

The proposed system architecture of Section 5.2 consists of several concurrently operating subsystems, the internal resources. An enticing approach would be to generate local supervisors for the individual internal resources such that the desired goal is fulfilled. This problem has been studied by Willner (1991) who shows that it can be solved, if and only if the specification satisfies a certain separability property. Essentially, this property means that the specification can be written as the synchronous composition of a number of transition machines with alphabets equal to the alphabets of the respective resources. Disregarding the fact that Willner (1991) is concerned with languages, and consequently deterministic automata (though an algorithm for checking the separability property involves generating nondeterministic automata), the separability property is in general not satisfied by the collection of product routes arising in our systems.

To guarantee a minimally restrictive and nonblocking supervisor, it seems that we will have to compose all the individual product routes in parallel. Now, the product routes can be seen as DEPs sharing a set of mutual resources. Since the number of sharing processes, that is, the number of concurrently running products, is not known beforehand, we will model their sharing of the mutual resources by *interleaving*. Arguments for this has already been given in Section 2.5.1. See also Hoare (1985).

In our interpretation of Petri nets, interleaving is inherent in the concurrent execution of two or more nets. At each time instant any net can execute its own event, totally asynchronously with any of the other nets. Thus, at each time instant the total system behaves as either of the nets completely non-deterministically, but at no time will two nets engage in the same action synchronously. This generates a language of the total system that is the interleaving of the languages of the respective nets. For bounded Petri nets, it is also easy to verify that this indeed represents the transition machine resulting

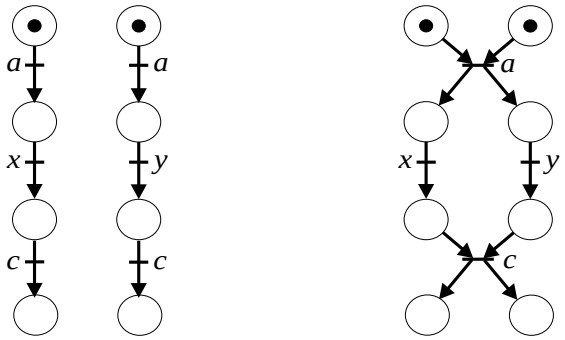



**Figure 5.20:** The interleaving (left) and the synchronous composition (right) of two Petri nets.

from interleaving the transition machine representations of the original Petri nets. See Figure 5.20, left. Interleaving is also inherent in a single Petri net, given that the net contains multiple tokens and at least two different transitions labeled by the same events. This is illustrated by Figure 1.5 on page 8.

The synchronous composition of two DEPs require simultaneous participation of both of the processes involved. These events are (usually) those that are present in the alphabets of both processes; events in the alphabet of one process not present in the other's alphabet can, of course, be executed freely. The synchronous composition of two Petri nets is effected by "layering" transitions with equal labels on top of each other, see Figure 5.20, right. This is known as the act of *fusing transitions*. Fused transitions can fire if and only if both nets are ready to fire them simultaneously. When connected events are present, we have to fuse more than two transitions. For instance, a transition labeled by an event $\langle y_1, a_1 \rangle$ in some product route, must be fused both with the transition labeled by $y_1$ of mover **R1**, and the transition labeled by $a_1$ of producer **M1**. See Figure 6.7.

For supervisor synthesis we generate the *local specification* by interleaving the product routes. The local specification is called the *interleaved synchronizable master recipe*, by Tittus (1995a); a more descriptive name, perhaps, albeit a lot longer. The term *local*, is used to denote that the specification's alphabet is not equal to the plant alphabet. The local specification does not necessarily include all events of the plant alphabet. Events pertaining to resources not used by any product are not in the alphabet of the local specification. As mentioned above, the connected events are not present within the plant at all.

The local specification represents the interleaving of all desired product routes. However, in the physical system, not all of these interleavings are possible. The local specification may, for instance, include a sequence of events that represents loading a producer two times in a row with no unloading in-between. If this producer cannot handle multiple products, this is not feasible in practice. To remove such sequences, we synchronize the local specification with the plant. In a Petri net representation this is done by fusing equally labeled transitions of the specification and the plant. As mentioned above, the transitions of the specification labeled by connected events are fused with two transitions of the plant. This generates the *global specification*, which represents all physically possible and desired paths for these product routes through this plant.

The global specification may not be satisfactory for supervisor synthesis, though. The theory of supervisor synthesis presented in the previous chapters assumed that the alphabet of the specification was equal to the plant alphabet, and that the specification refines the plant. These assumptions are not necessarily satisfied by the global specification. For one thing, the connected events have to be unfolded to become singular events. This cannot be done in the Petri net representation, though, since this would introduce exactly such "dangerous" places that we claimed was the reason for introducing connected events in the first place; see Example 5.3. However, if the transition machine representation of the bounded Petri net representing the global specification is generated *before* the connected events are unfolded, then states representing tokens in the "dangerous" places will not arise.

Unfolding the connected events in the transition machine representation of the global specification makes its alphabet equal to the alphabet of the plant. Events not included in the local specification, have been introduced by the synchronization with the plant. Also, the event sequences specified by the connected events cannot be interleaved by other events. The result of this is exactly the same as not introducing connected events at all into the product routes, interleaving the product routes, generating the full synchronous composition of the plant and the interleaved product routes, and then *remove all transitions that violate the connected events*. Thus, the transition machine representation of the global specification with the connected events unfolded is a subprocess of the full synchronous composition of the plant and the interleaved product routes without connected events. Therefore, it is clear that the global specification, as described above, with the connected events unfolded refines the plant. And so, the assumptions made in Section 4.3 are satisfied, meaning that the presented supervisory control theory for non-deterministic specification can be used. (We take accessibility for granted.)

Furthermore, there may exist *product unspecific* specifications that we always want the plant to satisfy during production of any type of products. Two producers may not be allowed to process simultaneously for instance. Maybe this draws to much power, and the electric account is overdrawn. Such product unspecific specifications can be introduced by removing states and transitions from the transition machine describing the global specification, thus generating a subprocess of the global specification. Introducing such specifications and unfolding the connected events generates the *final specification*. It is from this specification that we will synthesize the supervisor.

## 5.5 Applying the Theory

In this section we will apply the theory presented in the previous chapters to the models and systems described in the previous sections of this chapter. A summary of the method is shown in Figure 5.21. To generate the product routes from the operation lists, the method of Tittus (1995a) generates Petri nets that are essentially synchronized with a connectivity model of the plant. This generates the product routes in a high-level form. Using the synchronization model of the resources, see Figure 5.6, product routes labeled by connected events are generated. These are given as DEPs, forming *individual* specifications on the plant behavior, each only specifying the behavior of a subset of the plant resources. The product routes typically come in the form of bounded Petri nets. It is

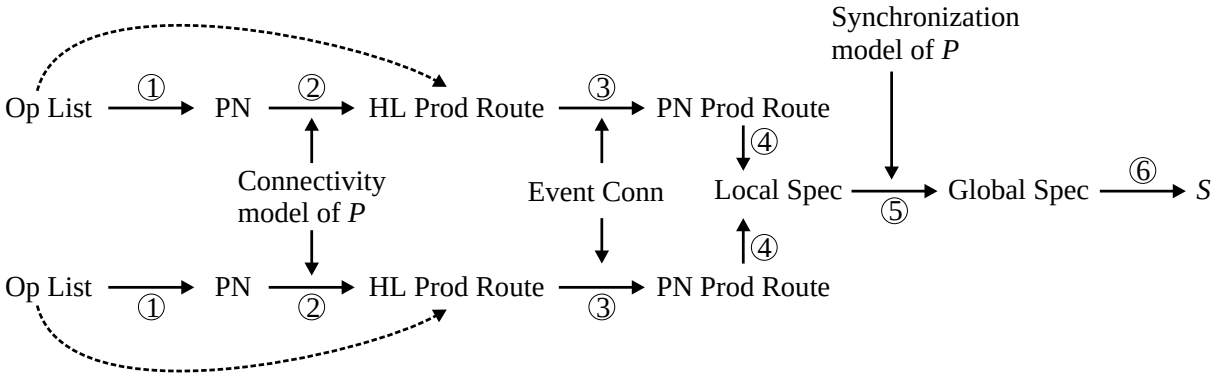



**Figure 5.21:** Summary of the method applied to generate a supervisor for a discrete event fabrication process. In Section 5.4 we took the shortcut indicated by the dotted arrows.

significant that the product routes include marked states, if no else, then the terminating states are marked. The product routes are to evolve asynchronously with respect to each other, though synchronously with respect to the plant, sharing the resources among them. Therefore, interleaving is used as the composition mechanism for the individual specifications. We generate the local specification,

$$Sp = S_1 \|_\emptyset \; S_2 \|_\emptyset \; \ldots \|_\emptyset \; S_m \tag{5.1}$$

for $m$ product routes $S_i$. Since any $S_i$ may be non-deterministic, and since they may contain mutual events $Sp$ becomes non-deterministic. Of course, the alphabet $\Sigma_{Sp} = \Sigma_{S_1} \bigcup \Sigma_{S_2} \bigcup \ldots \bigcup \Sigma_{S_m}$, and it contains only events from the plant, although some of these may be connected.

The synchronization model of the plant is the full synchronous composition of the synchronization models of the resources. That is,

$$P = P_1 \| \; P_2 \| \ldots \| P_n \tag{5.2}$$

for $n$ resources $P_j$. The alphabet becomes $\Sigma_P = \Sigma_{P_1} \bigcup \ldots \bigcup \Sigma_{P_n}$. Since each $P_j$ is deterministic, the plant, $P$, will also be.

We have claimed that the resources have disjoint alphabets, and this is typically the case when using generic reusable models. For instance, the models for supervisor synthesis shown in Figure 5.5 have this property. This is no requisite, however. The important thing is that the plant must be deterministic. There may occur situations when two resources need to share some "subresource". Like a machining tool that is common for all machines. This can be included in the presented models by introducing events that are common to the events of the subresource model, and thus synchronization between a resource and the subresource can be achieved. Note though, that this implies that for each resource that

is to share the subresource, there exists transitions labeled by certain events pertaining to that resource only, in the subresource. In fact, other than being deterministic, there are no other restrictions on the structure of the plant.

To yield the global specification, the composition $P \times Sp$ is obtained. Here we introduce the operator $\cdot \times \cdot$, which we will not define very formally. What we mean is that an edge $(q, \langle y, a \rangle, q')$ of $E_{Sp}$ labeled by a connected event $\langle y, a \rangle$ is synchronized with the edge $(p_1, y, p_1')$ as well as with the edge $(p_2, a, p_2')$, that both exist in $E_P$. The result is $(p_1 p_2 q, \langle y, a \rangle, p_1' p_2' q') \in E_{P \times S}$. Compare the synchronization vectors of Arnold (1994). Let $\mathcal{U}(\cdot)$ denote the operation of unfolding the connected events. Now we claim that we have the following subprocess relation $\mathcal{U}(P \times Sp) \leq P \| \mathcal{U}(Sp)$. That is, first synchronizing the plant and the local specification and then unfolding the connected events generates a subprocess of the synchronous composition of the plant and the local specification with unfolded events. This follows from our earlier claim that first unfolding the connected events of $Sp$, then synchronize $Sp$ with the plant under full synchronous composition, and finally remove all transitions violating the connected events is equivalent to $\mathcal{U}(P \times Sp)$. Since we in the first case remove transitions, it is obvious that we generate a subprocess of the synchronous composition of $P$ and $Sp$ with unfolded events. If the equality holds then it must also hold that $\mathcal{U}(P \times Sp) \leq P \| \mathcal{U}(Sp)$. From the remark to Theorem 2.48 we know that $P \| S$ always refines $P$. From Figure 2.5 we know that a subprocess refines its superprocess, so that $\mathcal{U}(P \times Sp)$ refines $P \| S$. Lemma 2.41 tells us that refinement is a transitive relation, so that $\mathcal{U}(P \times Sp)$ refines $P$.

Either $P \times Sp$ or $\mathcal{U}(P \times Sp)$ can be subject to product unspecific specifications that are introduced by removing states and/or transitions, thus generating the *final specification* $S \leq \mathcal{U}(P \times Sp) \leq P \| \mathcal{U}(Sp)$. Again we know that the final specification $S$ refines $P$. We also know that $\Sigma_S = \Sigma_P$, and we take accessibility of $S$ and $P$ for granted. Thus, the assumptions of Section 4.3 are met by $S$, so we can synthesize the supremal complete and trim subprocess of $S$, $\sup\left[\mathcal{C}(S) \bigcap \mathcal{A}(S) \bigcap \bar{\mathcal{A}}(S)\right] \equiv \sup \mathcal{CT}(S)$ by means of Algorithm 4.28. Since this is a subprocess of $S$ and since $P$ is deterministic, we know by Theorem 2.48 that $P \| \sup \mathcal{CT}(S) = \sup \mathcal{CT}(S)$, so that the closed-loop system is always nonblocking. Consequently, the supervisor guarantees that all products can be produced satisfactorily. Furthermore, with a deterministic plant and since $\sup \mathcal{CT}(S)$ refines $P$, so that $L(\sup \mathcal{CT}(S)) \subseteq L(P)$, we know from Lemma 3.22 that $\sup \mathcal{CT}(S)$ is inverse complete with respect to $P$. Therefore, the supervisor $\sup \mathcal{CT}(S)$ can act as a controller, generating commands and receiving responses. This controller can always drive the plant to produce the desired products.

## 5.6 Chapter Summary

In this chapter we have joined an approach to object-oriented modeling of discrete event fabrication processes with the supervisory control theory. Based on the use of interleaved product routes as specification for the behavior of the system, we synthesize a supervisor that guarantees that the closed-loop system is always nonblocking. Other, product unspecific, specifications can also be included. This forms a powerful framework for implementation of control systems for flexible manufacturing and assembly systems as well as batch processes. The object-oriented approach modularizes the control system according

to the structure of the physical system. Thus, the control system becomes intuitive and easy to understand. The modularity and the synthesis of control laws make the system adaptable to changes in its environment; new resources and new products can easily be included. Naturally, this also enhances the flexibility.

# Chapter 6

# Application Examples

To illustrate the concepts described in the previous chapters, we will in this chapter show three examples of implementing control for flexible fabrication processes. The first example details the assembly system used as the motivating example of Chapter 1. The second example illustrates the applicability of the proposed implementation scheme to chemical batch processes. This example was initially given by Tittus (1995c), and here we will focus on the controller synthesis, thus complementing the description of Tittus (1995c). Last, we will show an example of a manufacturing system, adapted from Zhou (1993), where the control for the described system is constructed manually, simultaneously with modeling the system. We will show that, given resource and product models, we can automatically construct the control of such a system. Naturally, we will also compare the control structure generated by Zhou (1993) and by our proposed method.

## 6.1 Assembly System

The model of the system Chapter 1, together with the product routes can be summarized as in Figure 6.1. These parts are shown separately in Figures 1.3 and 1.4 of Section 1.1. The global specification is shown in Petri net format in Figure 1.7 in the same section. Note that the synchronization models of Figure 5.6 are used. The state-machine representation of this global specification, assuming two parts each of **A**, **B**, **C** and **D**, is shown in Figure 6.2. Note that the whole state-machine is not shown, only a part relevant to the discussion on event connection and controller synthesis that follows. We can note that by first representing the global specification as a bounded Petri net, and then generating the state-machine representation we obtain the accessible states only. In that way we save a lot of states from the interleaving of the product routes, states that are not reachable after synchronization with the plant.

The global specification includes one marked state in which the evolution of the system is allowed to terminate. This state represents the completion of all products with the resources back in their initial states. This is typically the only marked state. However, with product routes including alternative paths there may exist several such states. It is not necessary, though, that all of these are marked. It may be the case that some of these paths are undesired. The cost of following them may be too high, for example. This marking and/or unmarking must, at present, be done manually. We will be looking at ways to specify such constraints, so that they can be included automatically.

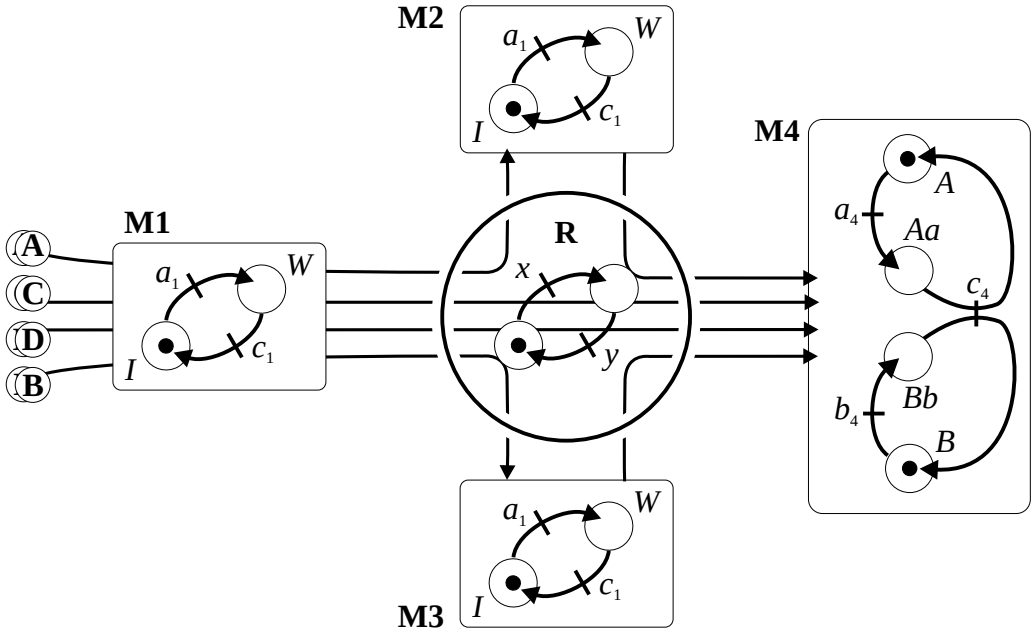



**Figure 6.1:** The product routes overlaid on the assembly system model. The robot **R** is shared among all product routes.

The connected events are specified as all physically possible exit-entry combination. We have full connectivity between all resources. However, for the described products only the connected events $\langle c_1, x\rangle$, $\langle c_2, x\rangle$, $\langle c_3, x\rangle$, $\langle y, a_2\rangle$, $\langle y, a_3\rangle$, $\langle y, a_4\rangle$ and $\langle y, b_4\rangle$ are relevant. Other combinations will not arise within the system with the specified product routes. Note that **M4** emits the finished products itself, the mover is not involved and no output station is modeled. The event $c_4$ represents "assembly finished and product emitted".

If the robot first loads a **C** part into **M4**, and then picks up a new **C** part, then the system deadlocks since there is nowhere for **R** to put the part it is holding. Similar scenarios for the other products also leads the system into a non-marked terminating state. The controller can prevent specific parts to enter the system, by disabling the event $a_1$ for those parts. The identity of the product to prevent from entering the system can be determined by the state-change that takes place for a specific $a_1$ event. We must assume that $a_1$ is controllable, otherwise there is no possibility to control the system, whatsoever. Should two **C** parts arrive in a row, then the system will inevitably deadlock. **M1** can only hold one part. Of course, the event $a_1$ represents a command issued by the controller to **M1**. Thus, **M1** itself would pick out the requested product, or choose among the allowed products.

Part of the resulting controller, with explanatory remarks, is shown in Figure 6.2. Instead of giving the state-names, we have labeled some transitions with both the event and the part pertaining to the event. This we can do once the global specification is generated. All transitions shown in Figure 6.2 can be continued into the marked state, which represents the completion of all products with all resources back in their initial states.

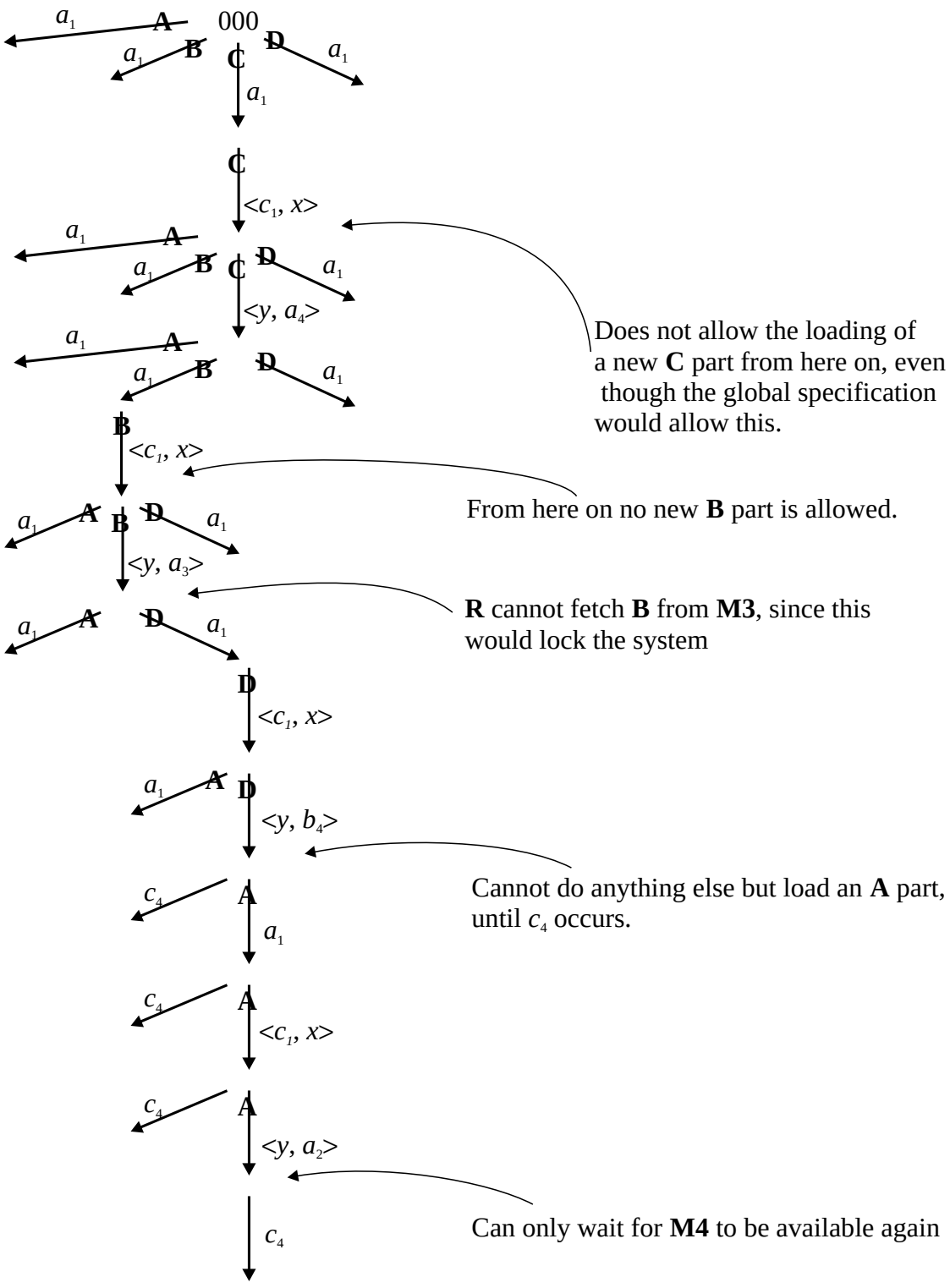



**Figure 6.2:** Part of the controller of Figure 1.7. Instead of naming the states, the part pertaining to a specific event is indicated by **A**, **B**, **C** and **D**.

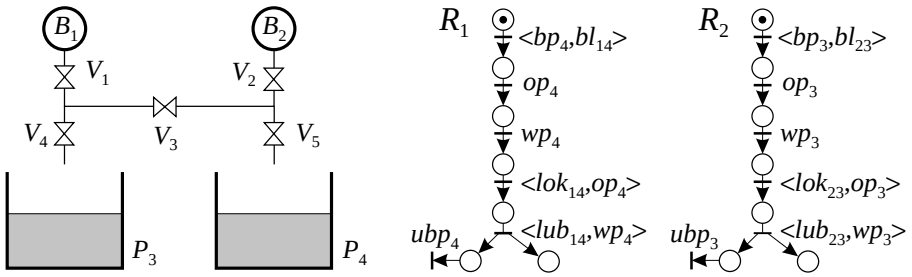



**Figure 6.3:** Plant and two recipes. Recipe $R_1$ describes filling of the processor $P_4$ from the buffer tank $B_1$, while $R_2$ describes filling of $P_3$ from $B_2$.

## 6.2 Batch Process

In Tittus (1995d) and Tittus (1995c) is shown an example of applying the supervisory control theory, as described in this thesis, to control and coordinate recipes in a chemical batch process. For such applications, Tittus (1995b) describes reusable automata-based modules, for modeling the plant and the products. Two types of resources have been defined, *processors*, such as tanks and reactors, and *transporters*, such as valves and pumps. Their equivalents for manufacturing systems, producers and movers, have been briefly described in Section 5.2. The recipes, describing the desired operations to be carried out by the processors, are given in a high-level graphical form. Again, this is very similar to the high-level product descriptions shown in Section 5.3, though specifically aimed at batch processes. The plant-unspecific high-level product descriptions, the *general recipes*, are mapped onto the plant to result in plant-specific *master recipes*. This is the static resource allocation shown in Example 5.2 on page 156, which is described in detail in Tittus (1995a) and Tittus (1995c).

We will not go into details of the plant modeling or the generation of the master recipes. The reader is referred to Tittus (1995c) for details. We merely present an example to show that the theory presented in this thesis is applicable, not only to manufacturing systems, but also to batch processes. The presented example is Example 4.17 of Tittus (1995c), though we have slightly tilted the focus to the generation of the controller.

We will assume that a suitable model of the plant and the product routes[1] are given. See Figures 6.3 and 6.4. The two recipes $R_1$ and $R_2$ describe the filling of the processors $P_3$ and $P_4$ from the buffer tanks $B_1$ and $B_2$. The filling of $P_4$ from $B_1$ requires that valves $V_1$, $V_3$ and $V_5$ are open, and that valves $V_2$ and $V_4$ are closed. In the same way, filling $P_3$ from $B_2$ requires that $V_2$, $V_3$ and $V_4$ are open, while $V_1$ and $V_5$ have to be closed. Obviously, this may lead to a deadlock, where neither of the recipes can be produced, since each closes some valve that the other requires to be open. In this small example, this could be avoided by having the recipes claiming the valves in the same order, so that the recipe that succeeds in claiming the first valve would have exclusive access to the other valves. However, in a more complicated (and more realistic) setting, this may not be possible, or at least it may be much too restrictive.

To handle the valve cooperation necessary for material transport in a batch process,

[1] The product routes are called *synchronizable master recipes*, by Tittus (1995c).

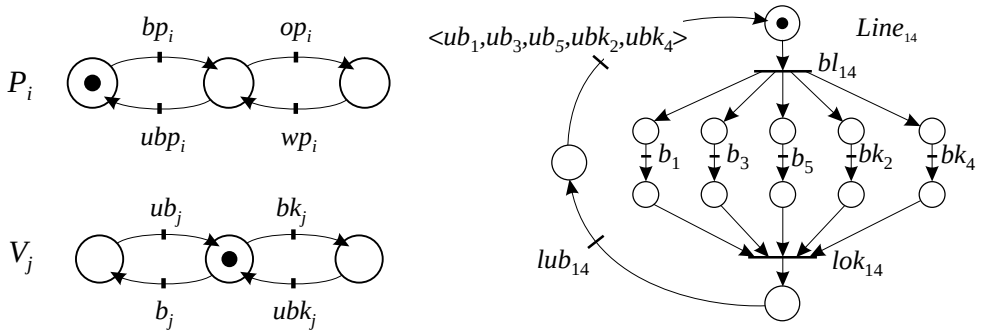



**Figure 6.4:** Generic models for processors, $P_i$ $(i = 1, 2)$ and valves, $V_j$ $(j = 1, 2, 3, 4, 5)$ to the left, and a connection-line connecting $B_1$ to $P_4$, to the right.

Tittus (1995c) generates *connection-lines*. These are abstract objects that have purely supervisory functions. For each possible connection between two processors, a connection-line can automatically be generated from the plant's topology. A connection-line serves as a high-level transport object for the recipes. In a manufacturing system a mover can hold the product as a whole, and the transfer is accomplished independently of any producer, that is, the producer is free as soon as the mover has fetched the product. In a batch process, on the other hand, the involved processors have to be kept unavailable for other activity while the material transfer takes place. The connection-lines provide this mutual exclusion, provided that the recipes hold on to the processors until the material transfer is finished.

A connection-line for the plant of Figure 6.3 is shown to the right in Figure 6.4. This connection-line connects the buffer tank $B_1$ with the processor $P_4$. The events of the connection-line represent booking and unbooking of the line, $bl_{14}$ and $lub_{14}$, respectively; confirmation that the line has been successfully booked, $lok_{14}$; and the booking, blocking, unbooking and unblocking of different valves. Compare the generic valve model given in Figure 6.4, lower left. Blocking a valve indicates that the valve is unbooked, but must not be opened by any other line until the valve is unblocked. The set of events marking the upper left transition of the connection-line, represent connected events.

The global specification is generated for the plant of Figures 6.3 and 6.4, in the same way as described above. The plant is synchronized under the synchronization constraint described by the recipes. There are two minor technicalities to consider here, both having to do with the connection-lines. For one, the connection-lines also define synchronization constraints that have to be considered when generating the plant model. These synchronization constraints are generated automatically, though, given the connectivity of the plant. There is no problem of including these when generating the plant model. Secondly, the connection-lines have to be interleaved with each other, though they are synchronized with both the valves of the plant and the recipes. The theory presented in Section 3.4 requires the plant to be deterministic, and the specification to have an alphabet equal to the plant alphabet. Regarding the connection-lines as specification clearly violates this, since there are events in the lines that are not present in the alphabet of the plant. These are the $bl_{ij}$ and the $lok_{ij}$ events. Regarding the connection-lines as part of the plant, also violates this, since then the plant is in general non-deterministic,

due to mutual connection-line events. Tittus (1995c) resolves this by re-labeling mutual connection-line events. The argument for this, is that the plant and the connection-lines are static. Once generated, they will not change, unless the plant is rebuilt. Therefore, the re-labeling does not have to be done more than once for each connection-line object. The result of the re-labeling is that the plant is deterministic. The reason not to re-label the events of the recipes, to generate a deterministic specification as well, is that the set of recipes is not static. It changes and evolves over time. New recipes are introduced, while old ones are finished and dismissed. Thus, there would have to be introduced in the plant new transitions labeled by events unique for each recipe. This would have to be done again and again, an issue of complexity we can do without. See also the discussion on multiple re-labeling contra interleaving in Section 2.5.1.

The resulting global specification is much to large to be presented here. The incidence matrix for the Petri net representation is shown by Tittus (1995c) on pages 141 and 142. It is a very sparse matrix, but it has 74 places and 38 transitions. The resulting controller automaton has 3645 states. Since all events are regarded as controllable, this is just the trim part of the global specification, always able to reach the state where both products have been produced, and the resources are back in their initial states. Some branches are removed, since some combinations of valve booking and blocking lead to deadlock of the system. Any situation in which both connection-lines succeed simultaneously to book/block one or more valves will inevitably lead to a deadlock. This is due to the fact that both connection-lines need access to all valves in order to be able to start operation. The resulting controller avoids this by only allowing one line to access the valves at a time.

## 6.3 Manufacturing System

In the approach of Zhou (1993) a "hybrid" top-down/bottom-up method is used to generate a Petri net model of the controlled, closed-loop behavior of manufacturing systems. This is done under a set of laws guaranteeing that the resulting system will have "nice" properties. These properties are that the closed-loop system will be

**Bounded**–no reachable marking is greater than some fixed vector $k$. This guarantees that no capacity overflows will occur; no resource will be requested to handle more products simultaneously, than it has the capacity to deal with.

**Live**–from any reachable marking, all transitions will eventually be enabled. This guarantees that all modeled processes can actually occur.

**Reversible**–from any reachable marking, the initial marking can always be reached. This implies a cyclic structure, so that the system can always be reinitialized from any reachable marking.

It is assumed that we always want the system to encompass these "nice" properties. However, this is not enough for the controlled system to *behave* "nicely". For instance, letting too many products enter the system simultaneously, can lead to a deadlock. Therefore, once the system is modeled, Zhou (1993) calculates the maximum number of products of

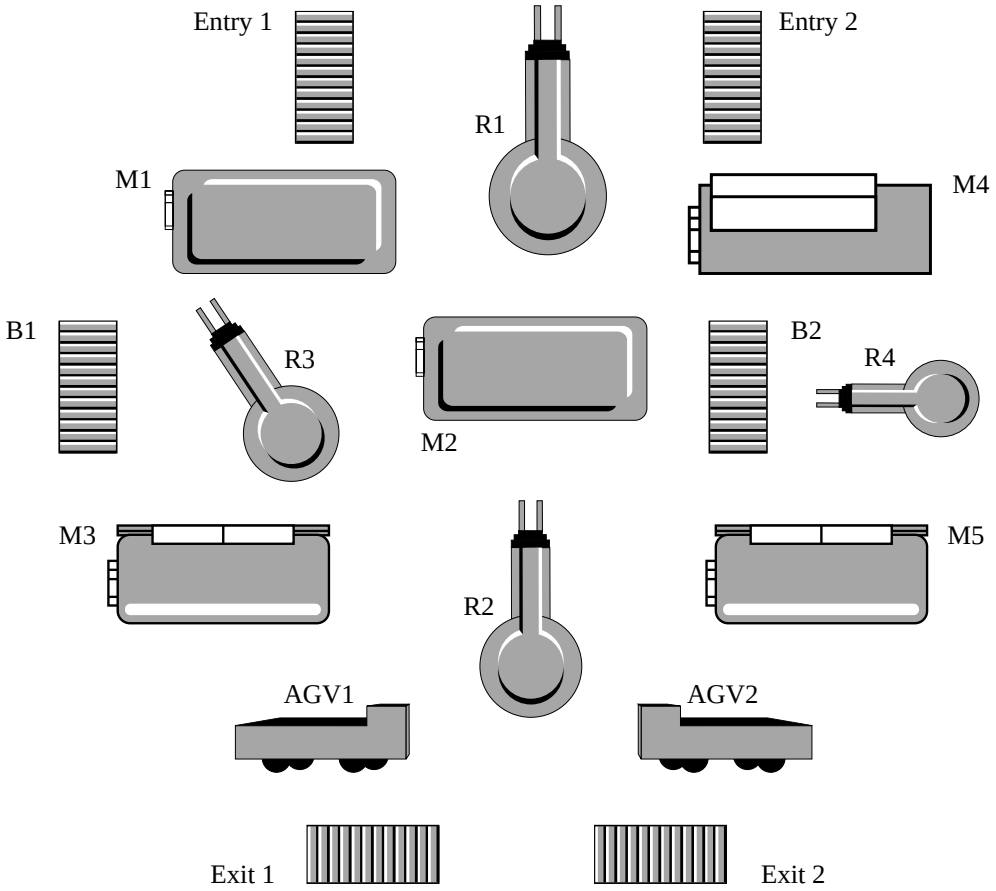



**Figure 6.5:** Layout of the automated manufacturing system of Chapter 6 of Zhou (1993).

each type that can be allowed to run concurrently through the system. Basically, this calculation determines the buffer capacity available for each type of product.

The example system given in Chapter 6 of Zhou (1993) is shown in Figure 6.5. Parts enter the system through the entry stations Entry 1 and Entry 2. The robot **R1** loads **M1**, **M2** and **M3** from the entry stations. Machines **M1** and **M2** are identical. From **M1** and **M2**, robot **R3** moves parts to the intermediate buffer **B1**, while **R4** unloads **M4** to the buffer **B2**. The buffers have capacity to hold $b_1$ number of F-parts for **B1**, and $b_2$ number of G-parts for **B2**. Robots **R3** and **R4** also load **M3** and **M5**, respectively, from the respective buffers **B1** and **B2**. Finally, **R2** unloads both **M3** and **M5**, loading the automatically guided vehicles **AGV1** and **AGV2**. Parts from **M3** are loaded onto **AGV1**, and parts from **M5** are loaded onto **AGV2**. The AGVs take different paths to their respective unloading stations, Exit 1 and Exit 2, from which the finished products disappear.

Two high-level product descriptions will be given for this system, the F and G operation lists shown in Figure 6.6. Both operation lists have the same structure, with one alternative operation that can be by-passed. This is typically a request for a location resource, a buffer, for intermediate storage, something which is not essential for the manufacturing of the product. It merely signifies that for these products intermediate storage

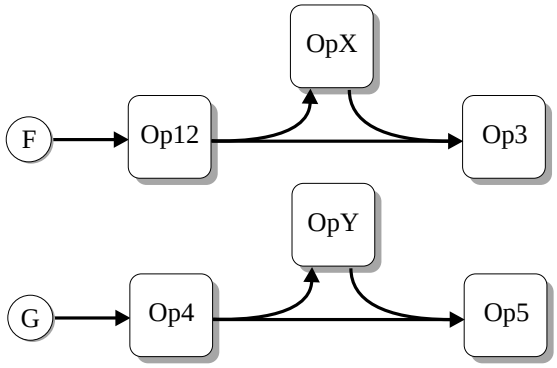



**Figure 6.6:** Operation lists for the two products defined for the system of Figure 6.5.

is allowed if resources that can meet the specified operations, *OpX* and *OpY*, respectively, are available. In this example we will assume that *OpX* can be met by **B1**, and *OpY* by **B2**, above.

In the specification of the physical system Zhou (1993) requires that the robots **R3** and **R4** should always move the parts through the intermediate buffers **B1** and **B2**. There are no clues as to *why* this is assumed to be necessary. However, assuming that **B1** and **B2** are FIFO-buffers, it would guarantee fairness to the processed parts. Otherwise, some parts could be held in the buffers, while other parts were transported directly from, say, **M2** to **M3** even though these other parts entered the system after the parts in the buffer. This could be resolved outside the controller, by the controller, but for comparison with the system of Zhou (1993), we will also make the assumption that there is no direct connectivity between **M1**/**M2** and **M3**, as well as between **M4** and **M5**. This means that the alternative to move the parts directly between the machines is lost in the product route.

We will assume that operation *Op12* can be performed by both **M1** and **M2**, and that operations *Op3*, *Op4* and *Op5* can be performed by **M3**, **M4** and **M5**, respectively. Thus, there arises an alternative path for the F-products, but not for the G-products. The product routes are shown gray in Figure 6.7. As usual, only the synchronization events are used. The events $a_i$, $x_i$, $v_i$, $b_{11}$ and $b_{21}$ denote booking of a producer, a robot, an AGV, **B1** and **B2** respectively. Similarly, the $c_i$, $y_i$, $w_i$,$b_{12}$ and $b_{22}$ events represent the releasing of the respective resource.

The connected events specified by the product routes are

- $\langle y_1, a_1\rangle$, $\langle y_1, a_2\rangle$ and $\langle y_1, a_3\rangle$ since **R1** loads all three machines **M1**, **M2** and **M3**;
- $\langle c_1, x_3\rangle$, $\langle c_2, x_3\rangle$, $\langle y_3, b_{11}\rangle$, $\langle b_{12}, x_3\rangle$ and $\langle y_3, a_3\rangle$, denoting that **R3** unloads both **M1** and **M2**, both loads and unloads **B1** and loads **M3** ;
- $\langle c_4, x_4\rangle$, $\langle y_4, b_{21}\rangle$, $\langle b_{22}, x_4\rangle$ and $\langle y_4, a_5\rangle$ since **R4** unloads **M4**, both loads and unloads **B2** and loads **M5** ;
- $\langle c_3, x_2\rangle$, $\langle c_5, x_2\rangle$, $\langle y_2, v_1\rangle$ and $\langle y_2, v_2\rangle$, denoting that **R2** unloads both **M3** and **M5** and loads both **AGV1** and **AGV2**.

The global specification, that is, the plant synchronized with the interleaved product routes, is shown in Figure 6.7. Compare this figure with Figure 6.9 of Zhou (1993). The gray places are the product routes and the white places represent the resources. Note that some superfluous places that would arise when the synchronization is performed automatically, have been omitted for clarity. When synchronizing two Petri nets, the number of places in the composed net is equal to the sum of the places of the synchronized nets. For every resource there would arise two places in the net of Figure 6.7. However, one of these places always holds redundant information, such as the place where **R1** is moving a product after an $x_1$ event. Removing these places makes the incidence matrix smaller, though the number of states described by the net is not altered. The net of Figure 6.7 is essentially the same as the net given by Zhou (1993) in Figure 6.9, though not cyclic.

The global specification is nondeterministic, as is easily verified. For instance, initially there is a choice of whether to let an F-part or a G-part to be moved by **R1**. This is in both cases modeled by the $x_1$ event, but leads to two different states, that is, markings. Such a nondeterministic choice can also arise in the usage of **R2**. Note that, Zhou (1993) does *not* consider events, only transitions. This makes the alphabet of the Petri net of Zhou (1993) equal to the transition set. Of course, all transitions are *named* differently, though some transitions can be labeled by the same event. Thus, the systems arising in the approach of Zhou (1993) are always deterministic. However, the controller experiences the plant through the generated events, so that for supervisory control we require the transitions to be labeled by events. This makes the global specification, as well as the controller, nondeterministic. Not labeling the transitions also makes it hard to automatically generate the global specification by synchronizing different modules. The systems of Zhou (1993) are, more or less, built manually, albeit under strict rules to guarantee the desired properties of boundeness, liveness and reversibility.

To generate the reachability graph, we will have to give some initial marking and define the buffer capacities $b_1$ and $b_2$. We will arbitrarily choose $b_1 = b_2 = 1$ and initially mark two F-parts and one G-part. The entire reachability graph, that is, the transition machine representation of the global specification, is much to large to be presented here. It has 530 states. A small portion of it is shown in Figure 6.8.

As can be seen from Figure 6.8, the system deadlocks in state 016. Since we have two F-parts, but **B1** only has capacity to hold one, when **B1** is full and **R3** picks up a new F-part from **M1** or **M2**, the system deadlocks. In fact, there are three scenarios that lead to the deadlock state 016. All three combinations of running the products through the system as $F_1F_2G$, $F_1GF_2$ and $GF_1F_2$, where the first F-part $F_1$ stays in **B1**, will deadlock the system. Of course, the controller algorithm will prune the transition machine so as not to include branches pertaining to these scenarios. Thus, the system will be *controlled* to exhibit a well-mannered behavior.

The approach of Zhou (1993) can also generate control-laws for the physical system by using the modeled closed-loop system. However, the correctness of the control-laws depends, not only on the structure of the Petri net, but also on the initial marking, that is, on how many products of each type are let into the system simultaneously. The calculation of these numbers is a crucial aspect for correct behavior of the control. As noted by Zhou (1993), this calculation is not straightforward to perform for general systems. It is suggested that the reachability graph be used for this. We argue that, if the reachability

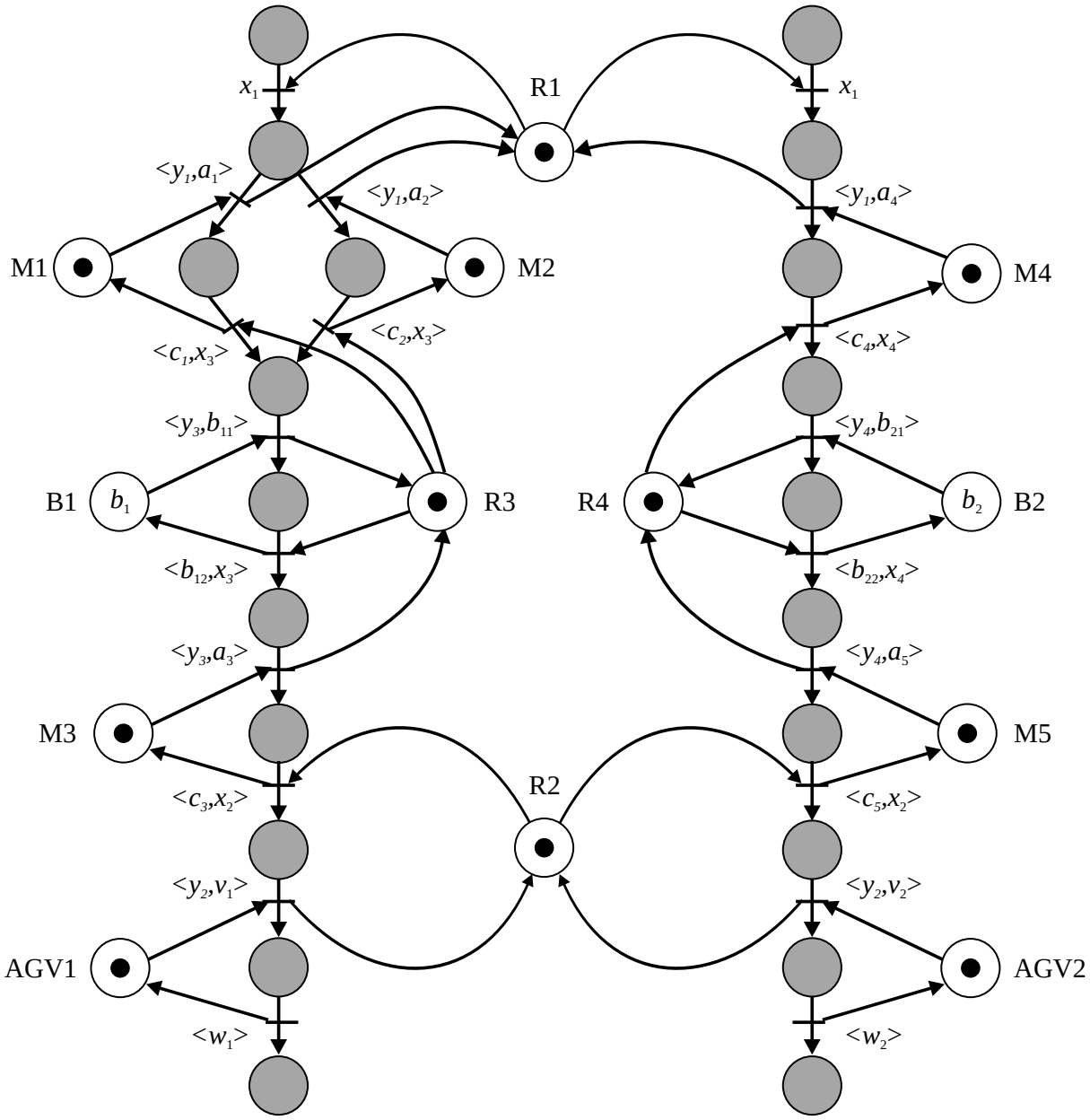



**Figure 6.7:** The global specification, $P \| (F \|_{\emptyset} G)$ under the synchronization constraint given by the specifications. The product routes $F$ and $G$ are shown in gray, while the places pertaining to the resources are white.

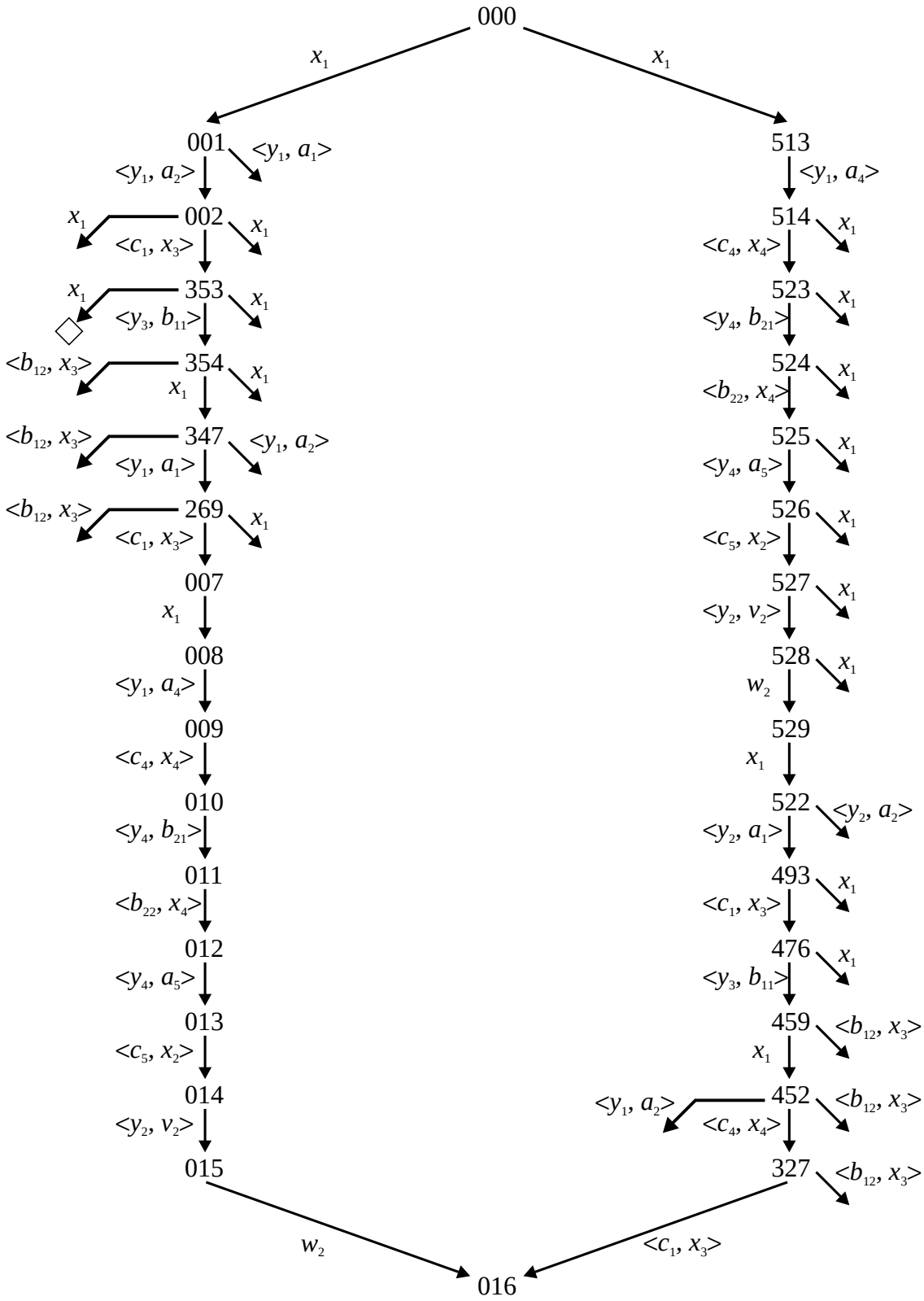



**Figure 6.8:** Part of the transition machine representation of the Petri net of Figure 6.7. The states have been arbitrarily numbered. 000 is the initial state. Transitions reaching non-numbered states can be continued in the total automaton. The rhomboid denotes a forbidden state. The marked state is not shown.

graph has to be used, then our method of generating control-laws is less restrictive than the approach of Zhou (1993).

As shown above, the controller inherently determines how many parts can be allowed to be operated upon simultaneously, regardless of the number of *requested* production units. If the robot **R1**, for instance, picks parts "intelligently", then there is no problem with deadlock, no matter how many F-tokens there are in the initial marking. Furthermore, it seems to us that when buffers are shared between products of different types, the number of products of each type that can be let into the system is not well-defined; see, for instance, point 2 on page 78 of Zhou (1993). The total number of all products is required to be less than or equal to a fixed value $b$. Of course, choosing the relative proportions between the products is a scheduling issue. But it also has a great impact on the control-laws. Our approach is to separate scheduling from control, generating a controller expressing control-laws un-influenced by the scheduling process. *When* a part to be produced arrives, we calculate *how* we can let this part through the system. Zhou (1993) guarantees that *if* the parts arrive in adequate order and numbers, *then* the system is able to let them through.

We can note also that the approach of Zhou (1993) does not seem to be able to handle the case of multiple movers that may get in the way of each other. For instance, it seems unlikely to us that **R1** and **R3**, say, could load and unload **M1** or **M2** simultaneously. The Petri net shown on page 135 of Zhou (1993) allows this. Markings such that $p_{11}$ and $p_{14}$ are simultaneously marked are not excluded. These places represent "**R1** acquiring a pallet from Entry 1 and loading **M1** or **M2**", and "**R3** unloading **M1** or **M2** and placing an intermediate F-part in Buffer 1", respectively (see page 129, Table 6.2 of Zhou (1993)). There seems to be no mentioning whatsoever of this problem in the book of Zhou (1993). On the contrary, for the example of Chapter 6 (page 123), the requirement is that "the system is not deadlocked, and has the smallest amount of starving", and "guaranteeing the net's boundedness, liveness and reversibility".

However, deadlock could arise from a circular wait by two movers wanting to enter some mutual area. In our approach, this can be included, and the controller would see to it that this situation would never arise. For instance, by specifying that all markings with both $p_{11}$ and $p_{14}$ simultaneously marked are forbidden states, this is achieved. In Figure 6.8 the small rhomboid reachable via $x_1$ from the state 353 marks such a state. In this state **R3** is unloading **M1** while **R1** loads **M1**. Marking this state as forbidden removes it from the controller, so that in state 353 the only allowed $x_1$ event is the one representing **R1** fetching a G-part. Of course, for the general case, where there are certain areas in the fabrication system from which movers have to be mutually excluded, these aspects have to be modeled. Then, the simple resource models used in this example do not suffice.

Another aspect of interest when regarding the work of Zhou (1993) is that Zhou does not consider events. Only unlabeled transitions are modeled. This means that, though Zhou (1993) interleaves the product routes, the resulting system is always deterministic since every transition is unique. Apart from the fact that non-determinism cannot arise, we believe that the absence of events, that is, transition labels, is the main reason for the approach of Zhou not to be suitable for automatic synthesis. Since the transitions are not labeled, how do we specify which transitions to synchronize? This is, in fact a critical issue, since, as Zhou (1993) says, when the physical system is altered "... the Petri net

model needs to be modified. Such modification may destroy the desirable properties." With our approach, when the physical system is altered, the model can be automatically generated for any type of product that is to be processed by the new system. Note also that Zhou does not use the term *interleaving* nor considers that it is the product routes that are essentially modeled.

Though the generated models are basically the same, the approach and purpose of generating them is different for Zhou (1993) and for us. The approach of Zhou (1993) is stated as

> *Given the specifications of a manufacturing system, model the system as a Petri net such that its structure and initial markings make it bounded, live and reversible.*
>
> Zhou (1993), Section 2.3

Then, Zhou (1993) claims, the resulting Petri net model can be translated to supervisory control code for actual execution of a manufacturing system.

Our approach can be stated as

> *Given a model of a fabrication process and a specification for its behavior, generate a control system such that the closed-loop system is nonblocking.*

To guarantee that $P \| S$ is nonblocking we know that $S$ has to be complete. The model is given as generic reusable resource models. However, even though each model itself possesses "nice" properties, they are live, bounded and reversible, their composition may not. Not unless either

1. the system is composed according to strict rules that preserve the properties, or
2. a controller is generated that prevents the system from exhibiting unwanted properties.

Of course, Zhou (1993) takes the first approach, while we take the second.

# Chapter 7

# Conclusions

In this thesis we have been concerned with control of a certain class of fabrication processes, namely processes that are such that their continuous behavior can be disregarded. The processes can therefore be described by transition machines. Typical examples of such processes are manufacturing and assembly systems, but it may also be adequate to model chemical batch processes on this level of abstraction.

We have proposed an object-oriented modeling approach for such systems, building on the ability to abstract the general behavior of certain classes of resources, machines, robots etc. The result of this is general models that can be adapted and reused for other implementations of similar systems. In this way the arrangement of the physical system is used to structure the control system coherently. To achieve reusability the removal of application-specific aspects is of importance. This can be achieved by separating the control of the individual subsystems from the control of the system as a whole. However, this also necessitates some, preferably automatic, way of binding the individual parts together.

In the supervisory control theory we have found the means to tie the modules of the control system together. Given that we have suitable models of the resources and the desired production sequences, we can synthesize control laws guaranteed to achieve the required production. The models suitable for this approach are surprisingly simple even for ample use in quite complex systems.

The specification for the global desired production sequences generally become non-deterministic, in the sense that the same sequence of events can lead to any of a number of states. Therefore, we have generalized the supervisory control theory to the case of non-deterministic systems. In doing so, we have also considered some aspects related to non-deterministic plants. We have succeeded in showing that the supervisory control theory can be used to generate control laws given non-deterministic specifications. We have not achieved utter generalization, though, since at present we are unable to show that our algorithms work for non-deterministic plant as well as specification.

All in all, we have in this thesis, together with the earlier work presented on page ii, shown how object-oriented modeling principles can be mated with the supervisory control theory to form a powerful framework for the implementation of flexible control systems for discrete event fabrication processes. The emphasis of this work has been on the theoretical side, focusing on the supervisor synthesis.

Some reviewers of the papers that preceded this thesis have taken the standpoint that

the main theorems and proofs are merely minor and obvious extensions to the supervisory control theory. The theorems may seem obvious once presented, but their derivation has by no means been straightforward. As for the theorems being a minor extension, they seem crucial for applying the supervisory control theory to real systems modeled by object oriented principles.

We have avoided the question of optimality. Our supervisor merely expresses all allowed paths through the system. How and when to choose what specific path is still an open question. We know though, that the paths allowed by the supervisor are the only paths worth considering when making such intricate decisions. Our models have also avoided resource failures, though the presented theory raises no such constraints. Given that we have the power to perform the calculations, the models can be as complex as necessary for the particular application.

Martin Fabian
Göteborg, August 10, 2000

# Appendix A

# Basic Set Results

In this appendix we show some basic set theoretic results that are used (mostly implictly) in the proofs. Some of these basic results will also be proven. These results are specific to the presented work, and this appendix is intentionally kept very terse. The reader is referred to more comprehensive literature for more elaborate explanations, such as Tremblay (1987).

Note that throughout this appendix, an exclamation mark in bold font, **!**, means that a contradiction to the initial assumptions has been reached.

## A.1 Singular Sets

In this section we will regard the sets $A$, $B$ and $C$ as subsets of some universal set $\Sigma$. The elements of $\Sigma$ are considered to be *singular* , that is, the elements are themselves *not* sets. The presented results hold equally for sets of sets, but we make the distinction, since in the next setion we will consider results specific to sets of sets; results that are not applicable to sets of singular elements.

For arbitrary sets $A, B, C \subseteq \Sigma$ , the following results hold.

**Lemma A.1**

$$A \bigcap B \subseteq C \Leftrightarrow A \bigcap B \subseteq B \bigcap C \tag{A.1}$$

**Lemma A.2**

$$A \subseteq C \Rightarrow A \bigcap B \subseteq B \bigcap C \tag{A.2}$$

**Lemma A.3**

$$A \bigcap B = A \Leftrightarrow A \subseteq B \Leftrightarrow A \bigcup B = B \tag{A.3}$$

## A.2 Mappings

Let $A$ and $B$ be arbitrary disjoint sets. Let $f : A \bigcup B \rightarrow 2^{\Sigma}$ be a function mapping elements of $A$ and $B$ into subsets of some set $\Sigma$. Define $f(A) = \bigcup_{\forall a \in A} f(a)$, and $F(A) = \bigcap_{\forall a \in A} f(a)$. Then the following relations hold. See also Figure A.1.

**Lemma A.4**

$$\forall a \in A \, \forall b \in B \; f(b) \subseteq f(a) \Leftrightarrow \forall a \in A \; f(B) \subseteq f(a) \tag{A.4}$$

**Proof.** ( $\Rightarrow$) Assume $\forall a \in A \; \forall b \in B \; f(b) \subseteq f(a)$ but $\exists a' \in A \; f(B) \not\subseteq f(a')$. Then $\exists \sigma \in f(B) \wedge \sigma \notin f(a') \Leftrightarrow \exists b' \in B \; \sigma \in f(b') \wedge \sigma \notin f(a') \Leftrightarrow f(b') \not\subseteq f(a')$ !

( $\Leftarrow$) Assume $\forall a \in A \; f(B) \subseteq f(a)$ but $\exists a' \in A \; \exists b' \in B \; f(b') \not\subseteq f(a')$. Then

$\exists \sigma \in f(b') \wedge \sigma \notin f(a') \Leftrightarrow \sigma \in f(B) \wedge \sigma \notin f(a') \Leftrightarrow f(B) \not\subseteq f(a')$ ! ■

**Lemma A.5**

$$\forall a \in A \; f(B) \subseteq f(a) \Rightarrow f(B) \subseteq f(A) \tag{A.5}$$

**Proof.** ($\Rightarrow$) Assume $\forall a \in A \; f(B) \subseteq f(a)$ but $f(B) \not\subseteq f(A)$. Then $\exists \sigma \in f(B) \wedge \sigma \notin f(A) \Leftrightarrow \sigma \in f(B) \wedge \forall a' \in A \; \sigma \notin f(a') \Leftrightarrow f(B) \not\subseteq f(a')$ !

($\not\Leftarrow$) Assume $f(B) \subseteq f(A)$ but $\exists a' \in A \; f(B) \not\subseteq f(a')$ . Then $\exists \sigma \in f(B) \wedge \sigma \notin f(a')$. No contradiction, $\sigma$ can belong to $f(\cdot)$ of some other element in $A$.

■

**Lemma A.6**

$$f(B) \subseteq f(A) \Leftrightarrow \forall b \in B \; f(b) \subseteq f(A) \tag{A.6}$$

**Proof.** ($\Rightarrow$) Assume $f(B) \subseteq f(A)$ but $\forall b' \in B \; f(b') \not\subseteq f(A)$. Then $\exists \sigma \in f(b') \wedge \sigma \notin f(A) \Leftrightarrow \sigma \in f(B) \wedge \sigma \notin f(A) \Leftrightarrow f(B) \not\subseteq f(A)$ !

($\Leftarrow$) Assume $\forall b \in B \; f(b) \subseteq f(A)$ but $f(B) \not\subseteq f(A)$. Then $\exists \sigma \in f(B) \wedge \sigma \notin f(A) \Leftrightarrow$ $\exists b' \in B$ such that $\sigma \in f(b') \wedge \sigma \notin f(A) \Leftrightarrow f(b') \not\subseteq f(A)$! ■

And here are the duals to the three lemmas above.

**Lemma A.7**

$$\forall a \in A \; \forall b \in B \; f(b) \subseteq f(a) \Leftrightarrow \forall b \in B \; f(b) \subseteq F(A) \tag{A.7}$$

**Proof.** ($\Rightarrow$) Assume $\forall a \in A \; \forall b \in B \; f(b) \subseteq f(a)$ but $\exists b' \in B \; f(b') \not\subseteq F(A)$. Then $\exists \sigma \in f(b') \wedge \sigma \notin F(A)$ then $\exists a' \in A$ such that $\sigma \notin f(a')$ so that $f(b') \not\subseteq f(a')$ !

($\Leftarrow$) Assume $\forall b \in B \; f(b) \subseteq F(A)$ but $\exists a' \in A \; \exists b' \in B \; f(b') \not\subseteq f(a')$. Then $\exists \sigma \in f(b') \wedge \sigma \notin f(a') \Leftrightarrow \sigma \in f(b') \wedge \sigma \notin F(A) \Leftrightarrow f(b') \not\subseteq F(A)$ ! ■

**Lemma A.8**

$$\forall b \in B \; f(b) \subseteq F(A) \Rightarrow F(B) \subseteq F(A) \tag{A.8}$$

**Proof.** ($\Rightarrow$) Assume $\forall b \in B\ f(b) \subseteq F(A)$ but $F(B) \not\subseteq F(A)$. Then $\exists \sigma \in F(B) \wedge \sigma \notin F(A) \Leftrightarrow \forall b \in B\ \sigma \in b \wedge \sigma \notin F(A)$. Thus $\forall b \in B\ f(b) \not\subseteq F(A)$ !

($\not\Leftarrow$) Assume $F(B) \subseteq F(A)$ but $\exists b' \in B\ f(b') \not\subseteq F(A)$. Then $\exists \sigma \in f(b') \wedge \sigma \notin F(A)$. No contradiction. There may exist $b'' \in B$ such that $\sigma \notin f(b'')$ and then $\sigma \notin F(B)$. ■

**Lemma A.9**

$$\forall b \in B\ f(b) \subseteq F(A) \Rightarrow f(B) \subseteq f(A) \tag{A.9}$$

**Proof.** ($\Rightarrow$) Assume $\forall b \in B\ f(b) \subseteq F(A)$ but $f(B) \not\subseteq f(A)$. Then $\exists \sigma \in f(B) \wedge \sigma \notin f(A) \Leftrightarrow \exists b' \in B\ \sigma \in f(b') \wedge \sigma \notin F(A)$ !

($\not\Leftarrow$) Assume $f(B) \subseteq f(A)$ but $\exists b' \in B\ f(b') \not\subseteq F(A)$. Then $\exists \sigma \in f(b') \wedge \sigma \notin F(A) \Leftrightarrow \sigma \in f(B) \wedge \exists a' \in A\ \sigma \notin f(a')$. No contradiction, $\sigma$ can belong to some other $a'' \in A$ ■

**Lemma A.10**

$$F(B) \subseteq F(A) \Leftrightarrow \forall a \in A\ F(B) \subseteq f(a) \tag{A.10}$$

**Proof.** ($\Rightarrow$) Assume $F(B) \subseteq F(A)$ but $\exists a' \in A\ F(B) \not\subseteq f(a')$. Then $\exists \sigma \in F(B) \wedge \sigma \notin f(a') \Leftrightarrow \sigma \in F(B) \wedge \sigma \notin F(A)$ !

($\Leftarrow$) Assume $\forall a \in A\ F(B) \subseteq f(a)$ but $F(B) \not\subseteq F(A)$. Then $\exists \sigma \in F(B) \wedge \sigma \notin F(A) \Leftrightarrow \sigma \in F(B) \wedge \exists a' \in A\ \sigma \notin f(a')$ ! ■

**Lemma A.11**

$$\forall a \in A\ \forall b \in B\ f(b) \subseteq f(a) \Leftrightarrow f(B) \subseteq F(A) \tag{A.11}$$

**Proof.** ($\Rightarrow$) Assume $\forall a \in A\ \forall b \in B\ f(b) \subseteq f(a)$ but $f(B) \not\subseteq F(A)$. Then $\exists \sigma \in f(B) \wedge \sigma \notin F(A) \Leftrightarrow \exists b' \in B\ \sigma \in b' \wedge \exists a' \in A\ \sigma \notin a'$. Then, of course, $f(b') \not\subseteq f(a')$ !

($\Leftarrow$) Assume $f(B) \subseteq F(A)$ but $\exists a' \in A\ \exists b' \in B\ f(b') \not\subseteq f(a')$. Then $\exists \sigma \in f(b') \wedge \sigma \notin f(a') \Leftrightarrow \sigma \in f(B) \wedge \sigma \notin F(A)$ ! ■

The following lemma proves an important equality concerning conformity.

**Lemma A.12**

$$\forall a', a'' \in A\ f(a') = f(a'') \Leftrightarrow \forall a \in A\ f(a) = f(A) \Leftrightarrow \forall a \in A\ f(a) = F(A) \tag{A.12}$$

**Proof.** We will only prove the first equivalence, the second follows similarily (and intuitively).

($\Rightarrow$) Assume $\forall a', a'' \in A\ f(a') = f(a'')$ but $\exists a''' \in A\ f(A) \neq f(a''')$ !

($\Leftarrow$) Assume $\forall a \in A\ f(A) = f(a)$ but $\exists a', a'' \in A\ f(a') \neq f(a'')$. Then $\exists \sigma \in f(a') \wedge \sigma \notin f(a'')$ ! ■

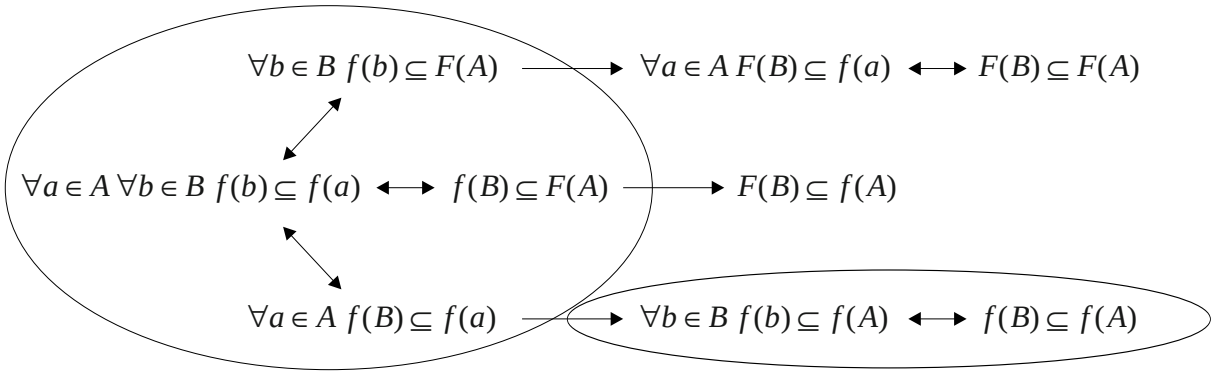



**Figure A.1:** The basic relations shown by Lemma A.4 to Lemma A.11. The left oval encircles the equivalent abstract expressions representing (inverse) completeness, while the right oval encircle the equivalent abstract expressions representing (nverse) controllability.

## A.3 Sets of Sets

For a set of set of subsets $A \subseteq 2^{\Sigma}$ of some universal set $\Sigma$, the following results hold.

## A.4 Set Differences

Let $A, B, C \subseteq \Sigma$ be arbitrary sets of some universal set $\Sigma$, and let $\neg A$ denote the *complement* of $A$, that is $A \bigcup \neg A = \Sigma$. Let $A - B$ denote the *set difference* of $A$ and $B$, that is, $A - B$ is the set of elements of $A$ not in $B$. Furthermore, let the descending order of precedence be $\bigcap$, $\bigcup$ and $-$. Then the following holds.

**Lemma A.13**

$$A - B \bigcap C = (A - B) \bigcup (A - C) \tag{A.13}$$

**Lemma A.14**

$$A \bigcup B - C = (A - C) \bigcup (B - C) \tag{A.14}$$

**Lemma A.15**

$$A \bigcap (B - C) = A \bigcap B - C \tag{A.15}$$

# Appendix B

# Proof of Lemma 2.47

Lemma 2.47 states that for any three state machines $P$, $Q$, $R$, the $\cdot \parallel_A \cdot$ operator is associative, that is

$$P \parallel_A (Q \parallel_B R) = (P \parallel_A Q) \parallel_B R, \tag{B.1}$$

whenever $A = B$ or $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$.

Let $A - B$ denote the *set difference* of $A$ and $B$, that is, $A - B$ is the set of elements of $A$ not in $B$, and let the descending order of precedence be $\bigcap$, $\bigcup$ and $-$.

**Proof.** By definition, we have for $Q \parallel_B R$

$$\begin{array}{l} \forall(\langle q, r\rangle, \sigma, \langle q', r'\rangle) \in E_{Q \parallel_B R} \\ \quad \begin{array}{llll} \sigma \in B & \wedge (q, \sigma, q') \in E_Q & \wedge (r, \sigma, r') \in E_R & \vee \\ \sigma \in \Sigma_R - B & \wedge (q, \sigma, q') \in E_Q & \wedge \quad r = r' & \vee \\ \sigma \in \Sigma_R - B & \wedge \quad q = q' & \wedge (r, \sigma, r') \in E_R & \end{array} \end{array} \tag{B.2}$$

and we have for $P \parallel_A (Q \parallel_B R)$

$$\begin{array}{l} \forall(\langle p, q, r\rangle, \sigma, \langle p', q', r'\rangle) \in E_{P \parallel_A (Q \parallel_B R)} \\ \quad \begin{array}{llll} \sigma \in A & \wedge (p, \sigma, p') \in E_P & \wedge (\langle q, r\rangle, \sigma, \langle q', r'\rangle) \in E_{Q \parallel_B R} & \vee \\ \sigma \in \Sigma_P - A & \wedge (p, \sigma, p') \in E_P & \wedge \quad \langle q, r\rangle = \langle q', r'\rangle & \vee \\ \sigma \in \Sigma_Q \bigcup \Sigma_R - A & \wedge \quad p = p' & \wedge (\langle q, r\rangle, \sigma, \langle q', r'\rangle) \in E_{Q \parallel_B R} & \end{array} \end{array} \tag{B.3}$$

Combining these expressions we get

$$\begin{array}{l} \forall(\langle p, q, r\rangle, \sigma, \langle p', q', r'\rangle) \in E_{P \parallel_A (Q \parallel_B R)} \\ \quad \begin{array}{llllll} \sigma \in A \bigcap B & \wedge (p, \sigma, p') \in E_P & \wedge (q, \sigma, q') \in E_Q & \wedge (r, \sigma, r') \in E_R & \vee \\ \sigma \in A \bigcap (\Sigma_Q - B) & \wedge (p, \sigma, p') \in E_P & \wedge (q, \sigma, q') \in E_Q & \wedge \quad r = r' & \vee \\ \sigma \in A \bigcap (\Sigma_R - B) & \wedge (p, \sigma, p') \in E_P & \wedge \quad q = q' & \wedge (r, \sigma, r') \in E_R & \vee \\ \sigma \in \Sigma_P - A & \wedge (p, \sigma, p') \in E_P & \wedge \quad q = q' & \wedge \quad r = r' & \vee \\ \sigma \in (\Sigma_Q \bigcup \Sigma_R - A) \bigcap B & \wedge \quad p = p' & \wedge (q, \sigma, q') \in E_Q & \wedge (r, \sigma, r') \in E_R & \vee \\ \sigma \in (\Sigma_Q \bigcup \Sigma_R - A) \bigcap (\Sigma_Q - B) & \wedge \quad p = p' & \wedge (q, \sigma, q') \in E_Q & \wedge \quad r = r' & \vee \\ \sigma \in (\Sigma_Q \bigcup \Sigma_R - A) \bigcap (\Sigma_R - B) & \wedge \quad p = p' & \wedge \quad q = q' & \wedge (r, \sigma, r') \in E_R & \end{array} \end{array} \tag{B.4}$$

By definition we also have that $B \subseteq \Sigma_Q \bigcap \Sigma_R$ and that $A \subseteq \Sigma_P \bigcap (\Sigma_Q \bigcup \Sigma_R)$.

In the same way we have for $(P \parallel_A Q) \parallel_B R$

$$
\begin{array}{l}
\forall(\langle p,q,r\rangle, \sigma, \langle p',q',r'\rangle) \in E_{(P\parallel_A Q)\parallel_B R} \\
\quad \sigma \in A \bigcap B \qquad\qquad \wedge (p,\sigma,p') \in E_P \quad \wedge (q,\sigma,q') \in E_Q \quad \wedge (r,\sigma,r') \in E_R \quad \vee \\
\quad \sigma \in (\Sigma_P \bigcup \Sigma_Q - B) \bigcap A \qquad \wedge (p,\sigma,p') \in E_P \quad \wedge (q,\sigma,q') \in E_Q \quad \wedge \quad r = r' \quad \vee \\
\quad \sigma \in B \bigcap (\Sigma_P - A) \qquad \wedge (p,\sigma,p') \in E_P \quad \wedge \quad q = q' \quad \wedge (r,\sigma,r') \in E_R \quad \vee \\
\quad \sigma \in (\Sigma_P \bigcup \Sigma_Q - B) \bigcap (\Sigma_P - A) \quad \wedge (p,\sigma,p') \in E_P \quad \wedge \quad q = q' \quad \wedge \quad r = r' \quad \vee \\
\quad \sigma \in B \bigcap (\Sigma_Q - A) \qquad \wedge \quad p = p' \quad \wedge (q,\sigma,q') \in E_Q \quad \wedge (r,\sigma,r') \in E_R \quad \vee \\
\quad \sigma \in (\Sigma_P \bigcup \Sigma_Q - B) \bigcap (\Sigma_Q - A) \quad \wedge \quad p = p' \quad \wedge (q,\sigma,q') \in E_Q \quad \wedge \quad r = r' \quad \vee \\
\quad \sigma \in \Sigma_R - B \qquad\qquad \wedge \quad p = p' \quad \wedge \quad q = q' \quad \wedge (r,\sigma,r') \in E_R
\end{array}
\tag{B.5}
$$

By definition we now have that $A \subseteq \Sigma_P \bigcap \Sigma_Q$ and that $B \subseteq (\Sigma_P \bigcup \Sigma_Q) \bigcap \Sigma_R$.

For (B.4) and (B.5) above to be equal we have the verify that the following equalities hold for all $\Sigma_P$, $\Sigma_Q$ and $\Sigma_R$ under the given conditions for A and B, that $A = B$ or $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$.

$$
\begin{array}{rc}
1. & A \bigcap B = A \bigcap B \\
2. & A \bigcap (\Sigma_Q - B) = (\Sigma_P \bigcup \Sigma_Q - B) \bigcap A \\
3. & A \bigcap (\Sigma_R - B) = B \bigcap (\Sigma_P - A) \\
4. & \Sigma_P - A = (\Sigma_P \bigcup \Sigma_Q - B) \bigcap (\Sigma_P - A) \\
5. & (\Sigma_Q \bigcup \Sigma_R - A) \bigcap B = B \bigcap (\Sigma_Q - A) \\
6. & (\Sigma_Q \bigcup \Sigma_R - A) \bigcap (\Sigma_Q - B) = (\Sigma_P \bigcup \Sigma_Q - B) \bigcap (\Sigma_Q - A) \\
7. & (\Sigma_Q \bigcup \Sigma_R - A) \bigcap (\Sigma_R - B) = \Sigma_R - B
\end{array}
\tag{B.6}
$$

Let LHS denote the Left Hand Side of the expressions above, while RHS denotes the Right Hand Side.

1. LHS is obviously equal to RHS.

2. Note that it is always the case that $A \subseteq \Sigma_Q$. By Lemma A.15, LHS $= A \bigcap \Sigma_Q - B = A - B = A \bigcap (\Sigma_P \bigcup \Sigma_Q) - B$ =RHS. Thus LHS is equal to RHS under any configuration of A and B.

3. When $A = B$ LHS $= A \bigcap (\Sigma_R - A) = \emptyset = B \bigcap (\Sigma_P - B)$ =RHS. When $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$, LHS $= \Sigma_P \bigcap \Sigma_Q \bigcap \Sigma_R - \Sigma_Q \bigcap \Sigma_R = \emptyset = \Sigma_Q \bigcap \Sigma_R \bigcap \Sigma_P - \Sigma_P \bigcap \Sigma_Q$ =RHS.

4. When $A = B$, RHS $= (\Sigma_P \bigcup \Sigma_Q - A) \bigcap (\Sigma_P - A) = \Sigma_P - A$ =LHS. When $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$, RHS $= (\Sigma_P \bigcup \Sigma_Q - \Sigma_Q \bigcap \Sigma_R) \bigcap (\Sigma_P - \Sigma_P \bigcap \Sigma_Q)$. By Lemma A.14 and Lemma A.15, this is equal to $[(\Sigma_P - \Sigma_Q) \bigcup (\Sigma_P \bigcup \Sigma_Q - \Sigma_R) \bigcap (\Sigma_P - \Sigma_Q)$. Performing the intersection we get
$(\Sigma_P - \Sigma_Q) \bigcup [(\Sigma_P \bigcup \Sigma_Q - \Sigma_R) \bigcap (\Sigma_P - \Sigma_Q)]$. Now, since $(\Sigma_P \bigcup \Sigma_Q - \Sigma_R) \bigcap (\Sigma_P - \Sigma_Q) \subseteq (\Sigma_P - \Sigma_Q)$, we have $(\Sigma_P - \Sigma_Q) \bigcup [(\Sigma_P \bigcup \Sigma_Q - \Sigma_R) \bigcap (\Sigma_P - \Sigma_Q)]$ $=(\Sigma_P - \Sigma_Q)$ =LHS.

5. This is the dual to number 2. It is similarily verified that LHS $= B - A$ =RHS.

6. LHS $= (\Sigma_Q \bigcup \Sigma_R) \bigcap \neg A \bigcap \Sigma_Q \bigcap \neg B = \Sigma_Q \bigcap \neg A \bigcap \neg B = (\Sigma_P \bigcup \Sigma_Q) \bigcap \Sigma_Q \bigcap \neg A \bigcap \neg B =$ *RHS*. Again, this condition does not incur any constraints on A or B.

7. This is the dual to number 4. It is similarily verified that when $A = B$, LHS $= \emptyset =$RHS, and that when $A = \Sigma_P \bigcap \Sigma_Q$ and $B = \Sigma_Q \bigcap \Sigma_R$, LHS $= \Sigma_R - \Sigma_Q =$RHS.

Thus Lemma 2.47 is proven. ■

# Bibliography


Adiga (1993) — Adiga, S. (ed.) *Object-Oriented Software for Manufacturing Systems*, Chapman & Hall, 1993.

Adlemo (1995a) — Adlemo, A., S-A. Andréasson, M. Fabian, P. Gullander, B. Lennartson, *Towards a Truly Flexible Manufacturing System*, Computer Engineering Practice, Vol 3, No 4, 545-554, April 1995.

Adlemo (1995b) — Adlemo, A., P. Gullander, S-A Andréasson, M. Fabian, B. Lennartson, *Operator Control Activities: a Case Study of a Machining Cell*, 8th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'95, Beijing, China, October 1995.

Andréasson (1995) — Andréasson, S-A, A. Adlemo, M. Fabian, P. Gullander, B. Lennartson, *A Machining Cell Level Language for Product Specification*, 8th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'95, Beijing, China, October 1995.

Andréasson (1995) — Andréasson, S-A, A. Adlemo, M. Fabian, P. Gullander, B. Lennartson, *Database Design for Machining Cell Level Product Specification*, 21st IEEE Industrial Electronics Conference, IECON'95, Orlando, Florida, USA, November 1995.

Arnold (1994) — Arnold, A., *Finite Transition Systems*, Prentice-Hall International Series in Computer Science, 1994.

Azzopardi (1994) — Azzopardi, D., S. Lloyd, *Scheduling and Simulation of Multi Product Batch process Plant Through Petri Net Modeling*, 3rd International Intelligent Manufacturing Systems Symposium, IMS'94, Vienna, Austria, 1994.

Banaszak (1990) — Banaszak, Z. A., B. H. Krogh, *Deadlock Avoidance in Flexible Manufacturing Systems with Concurrently Competing Process Flows*, IEEE Transactions on Robotics and Automation, Vol. 6, No. 6, pp 724-734, 1990.

Balemi (1992) — Balemi, S., *Control of Discrete Event Systems: Theory and Application*, Ph.D. Thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1992.

Bauer (1991) Bauer, A., R. Bowden, J. Browne, J. Duggan, G. Lyons, *Shop Floor Control Systems. From Design to Implementation.*, Chapman and Hall, London, UK, 1991.

Brandt (1990) Brandt, R. D., V. Garg, R. Kumar, F. Lin, S. I. Marcus, W. M. Wonham, *Formulas for Calculating Supremal Controllable and Normal Sublanguages*, Systems and Control Letters 15, pp 111-117, 1990.

Cassandras (1993) Cassandras, C. G., *Discrete Event Systems: Modeling and Performance Analysis*, Aksen Associates Incorporated Publishers, 1993.

Cox (1986) Cox, B., *Object Oriented Programming, an Evolutionary Approach*, Addison-Wesley, 1986.

Eilenberg (1974) Eilenberg, S., *Automata, Languages and Machines*, Academic Press, 1974.

Fox (1992) Fox, M. S., *Towards a Common Sense Model of the Enterprise*, Plenary Session, 7th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology, INCOM'92, Toronto, Canada, 1992.

Giua (1991) Giua A., F. DiCesare, *Supervisory Design Using Petri Nets*, 30th Conference on Decision and Ccontrol, CDC'91, Brighton, England, 1991.

Gullander (1995) Gullander, P., M. Fabian, S-A Andréasson, B. Lennartson, A. Adlemo, *Generic Resource Models and a Message Passing Structure in an FMS Controller*, International Conference on Robotics and Automation, 1995 IEEE International Conference on Robotics and Automation, ICRA'95, Nagoya, Japan, May 1995.

Gullander (1995) Gullander, P., A. Adlemo, S-A Andréasson, M. Fabian, B. Lennartson, *Guidelines for Achieving Flexible Cell Control using Message-based Communication*, ESPRIT Conference on Integration in Manufacturing, IIM'95, September 1995.

Heymann (1991) Heymann, M., G. Meyer, *An Algebra of Discrete Event Processes*, NASA Technical Memorandum 102848, 1991.

Hoare (1985) Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

Hopcroft (1979) Hopcroft, J. E., J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Series in Computer Science, 1979.

Huang (1995) Huang, H.-H., F. L. Lewis, O. C. Pastravanu, A. Gürel, *Flow-Shop Scheduling Design in an FMS matrix Framework*, Control Engineering Practice, Vol. 3, No. 4, pp. 561-568, 1995.

Inan (1994) Inan, K., *Nondeterministic Supervision under Partial Observation*, Lecture Notes in Control and Information Sciences 199, Cohen and Quadrat (eds.), 39-48, Springer-Verlag, 1994.

Inan (1995) Inan, K., *On a Class of Timer Hybrid Systems Reducible to Finite State Automata*, Discrete Event Dynamic Systems: Theory and Applications, Vol. 5, 83-96, Kluwer Academic Press, 1995.

Jobling (1994) Jobling C. P., P. W. Grant, H. A. Barker, P. Townsend, *Object Oriented Programming in Control System Design: a Survey*, Automatica, Vol. 30, No. 8, 1221-1261, 1994.

Joannis (1992) Joannis, R., M. Krieger, *Object Oriented Approach to the Specification of Manufacturing Systems*, Journal of Computer Integrated Manufacturing Systems, Vol. 5, No. 2, 1992.

Krieger (1993) Krieger, M., *Multiactivity Paradigm for the Design and Coordination of FMSs*, Journal of Computer Integrated Manufacturing Systems, Vol 6, No 3, August 1993

Kumar (1991) Kumar, R., V. Garg, S. I. Marcus, *On Controllability and Normality of Discrete Event Dynamical Systems*, Systems and Control Letters 17, 157-168, 1991.

Kumar (1994) Kumar, R., M. A. Shayman, *Avoiding Blocking in Prioritized Synchronization Based Control of Nondeterministic Systems*, Lecture Notes in Control and Information Sciences 199, Cohen and Quadrat (eds.), 49-58, Springer-Verlag, 1994.

Kumar (1995) Kumar, R., V. Garg, *Modeling and Control of Logical Discrete Event Systems*, Kluwer Academic Publishers, 1995.

Lin (1988) Lin, F., A. F. Vaz, W. M. Wonham, *Supervisor Specification and Synthesis for Discrete Event Systems*, International Journal of Control, Vol. 48, No. 1, 321-332, 1988.

Meyer (1988) Meyer, B., *Object-oriented Software Construction*, Prentice-Hall International Series in Computer Science, 1988.

Milner (1989) Milner, R., *Communication and Concurrency*, Prentice-Hall International Series in Computer Science, 1989.

Overkamp (1994) Overkamp, A., *Supervisory Control for Nondeterministic Systems*, Lecture Notes in Control and Information Sciences 199, Cohen and Quadrat (eds.), 59-65, Springer-Verlag, 1994.

Overkamp (1995) Overkamp, A., *Control of Nondeterministic Discrete Event Systems using Failure Semantics*, European Control Conference, ECC'95, Rome, Italy, 1995.

Peterson (1981) Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc., 1981.

Ramadge (1987) Ramadge, P. J., W. M. Wonham, *Supervisory Control of a Class of Discrete Event Processes*, SIAM Journal of Control and Optimization, Vol. 25, No 1, 206-230, 1987.

Rolstadås (1992) Rolstadås, A., *One-of-a-kind Production*, Production, Planning and Control, Editorial, Vol. 3, No. 2, 117, 1992.

Sargent (1993) Sargent, P., *Inherently Flexible Cell Communications: A Review*, Journal of Computer Integrated Manufacturing, Vol. 6, No. 4, November, 1993.

Shayman (1994) Shayman, M. A., R. Kumar, *Supervisory Control of Nondeterministic Systems with Driven Events via Prioritized Synchronization and Trajectory Models*, Technical Report, University of Maryland, Maryland, USA, 1994.

Shlaer (1992) Shlaer, S., S. J. Mellor, *Object Lifecycles, Modeling the World in States*, Yourdon Press, 1992.

SP-88 Instrument Socciety of America, *Batch Control Systems: Models and Terminology*, Draft 9, 1994.

Tittus (1995a) Tittus, M., M. Fabian, *Automated Generation of Plant Specific Recipes in Batch Control*, ICCI'95, Hong Kong, June 1995

Tittus (1995b) Tittus, M., B. Egardt, B. Lennartson, *Plant and Product Models for Batch Processes*, European Control Conference, ECC'95, Rome, Italy, September 1995.

Tittus (1995c) Tittus, M., *Control Synthesis for Batch Processes*, Ph. D. Thesis, Technical Report No. 280, Control Engineering Laboratory, Chalmers University of Technology, G̈teborg, Sweden, October 1995.

Tittus (1995d) Tittus, M., M. Fabian, B. Lennartson, *Controlling and Coordinating Recipes in Batch Applications*, 34th IEEE Conference on Decision and Control, CDC'95, New Orleans, Louisiana, USA, December 1995.

Tremblay (1987) Tremblay, J. P., R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*, McGraw-Hill International Editions, Computer Science Series, 1987.

Truss (1991) Truss, J. K., *Discrete Mathematics for Computer Scientists*, Addison-Wesley International Computer Science Series, 1991.

Valckenaers (1994) Valckenaers, P., H. van Brussel, L. Bongaerts, *Programming, Scheduling and Control of Flexible Assembly Systems*, CIMIA'94, Rion Patras, Greece, June 1994

Willner (1991) Willner, Y., M. Heymann, *Supervisory Control of Concurrent Discrete Event Systems*, International Journal of Control, Vol. 54, No. 5, 1143-1169, 1991.

Wonham (1987) Wonham, W. M., P. J. Ramadge, *On the Supremal Controllable Sublanguage of a Given Language*, SIAM Journal of Control and Optimization, Vol. 25, No 3, 637-659, 1987.

Wonham (1988) Wonham, W. M., P. J. Ramadge, *Modular Supervisory Control of Discrete Event Systems*, Mathematics of Control, Signals and Systems, 1:13-30, 1988.

Zhou (1993) Zhou, M. C., F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers, 1993.